Learning Ansible

Use Ansible to configure your systems, deploy software,
and orchestrate advanced IT tasks

Madhurranjan Mohaan
Ramesh Raithatha

Learning Ansible

Use Ansible to configure your systems, deploy software, and orchestrate advanced IT tasks

Madhurranjan Mohaan

Ramesh Raithatha

[PACKT] open source*
PUBLISHING   community experience distilled

BIRMINGHAM - MUMBAI

Learning Ansible

Credits

Authors
Madhurranjan Mohaan
Ramesh Raithatha

Reviewers
Yves Dorfsman
Ajey Gore
Maykel Moya
Fernando F. Rodrigues
Patrik Uytterhoeven

Commissioning Editor
Edward Gordon

Acquisition Editor
Meeta Rajani

Content Development Editor
Ritika Singh

Technical Editors
Menza Mathew
Shruti Rawool

Copy Editors
Deepa Nambiar
Adithi Shetty

Project Coordinator
Judie Jose

Proofreaders
Stephen Copestake
Ameesha Green
Sandra Hopper

Indexers
Hemangini Bari
Monica Ajmera Mehta
Rekha Nair
Priya Sane
Tejal Soni

Graphics
Abhinash Sahu

Production Coordinator
Shantanu N. Zagade

Cover Work
Shantanu N. Zagade

About the Authors

Madhurranjan Mohaan is a passionate engineer who loves solving problems. He has more than 8 years of experience in the software industry. He worked as a network consultant with Cisco before starting his DevOps journey in 2011 at ThoughtWorks, where he learned the nuances of DevOps and worked on all aspects from Continuous Integration to Continuous Delivery. He is currently working at Apigee, where he enjoys dealing with systems at scale.

Madhurranjan has also worked with various tools in configuration management, CI, deployment, and monitoring and logging space, apart from cloud platforms such as AWS. He has been a presenter at events such as DevOpsDays and Rootconf, and loves teaching and singing. He is a first-time author and wishes to share his learning with readers from across the world.

Ramesh Raithatha is a DevOps engineer by profession and a cyclist at heart. He is currently working at Apigee and has worked on various facets of the IT industry, such as configuration management, continuous deployment, automation, and monitoring. He likes exploring stunning natural surroundings on two wheels in his leisure time.

Acknowledgments

About the Reviewers

Yves Dorfsman is a system administrator with experience in the oil and gas, financial, and software industries, both in traditional corporations and startups, supporting production and development environments.

Ajey Gore is the founder of CodeIgnition, a boutique DevOps consulting firm. Prior to starting CodeIgnition, he was the CTO for Hoppr and also the Head of Technology for ThoughtWorks. He has more than 15 years of experience and has worked with multiple technologies.

Ajey is well known in the tech community in India for DevOps, Cloud, Automation, Ruby, JavaScript, and Go languages. He organizes and runs the DevOpsDays India, RubyConf India, and GopherCon India conferences. He has a broad knowledge of configuration management tools, virtualization, cloud, and large-scale enterprise IT projects development.

Maykel Moya has been working in the field of systems and network administration since 1999. Previously, he worked at two of the largest ISPs in his hometown of Cuba, where he managed HA clusters, SAN, AAA systems, WAN, and Cisco routers. He entered the GNU/Linux landscape through Red Hat, but today, his main experience lies in Debian/Ubuntu systems. He relates to the "free software" philosophy.

Convinced through personal experience rather than human intervention that computer operations don't scale and are error-prone, he is constantly seeking ways to let software liberate people from tedious and repetitive tasks. With a Puppet background, he looked for alternatives and discovered Ansible in its early days. Since then, he has been contributing to the project.

Maykel is currently employed by ShuttleCloud Corp., a company that specializes in cloud data migration at scale. Here, he works as a site reliability engineer, ensuring that the machine fleet is always available and runs reliably. He also manages resources in an optimal manner. Ansible is one of the many technologies he uses to accomplish this on a daily basis.

Fernando F. Rodrigues is an IT professional with more than 10 years of experience in systems administration, especially with Linux and VMware. As a system administrator, he has always focused on programming and has experience in working on projects from the government sector to financial institutions. He is a technology enthusiast and his areas of interest include cloud computing, virtualization, infrastructure automation, and Linux administration.

Fernando is also the technical reviewer of the book, *VMware ESXi Cookbook*, *Packt Publishing*.

Patrik Uytterhoeven has over 16 years of experience in IT; most of this time was spent with Unix and open source solutions. At the end of 2012, he joined Open-Future, the first Zabbix reseller and training partner in Belgium, where he had the opportunity to certify himself as a Zabbix-certified trainer. Since then, he has been giving trainings and public demonstrations not only in Belgium, but around the world, such as in the Netherlands, Germany, Canada, Ireland, and so on. To make life easier, he made some Ansible roles for Red Hat/CentOS 6.x to deploy and update Zabbix. These roles and some others can be found in Ansible Galaxy at `https://galaxy.ansible.com/list#/users/1375`. At the moment, Patrik is writing a Zabbix-related Cookbook for Packt Publishing.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

To Amma (my mom), Appa (my dad), and my wife, Jai.

---Madhurranjan Mohaan

To my parents.

---Ramesh Raithatha

Table of Contents

Preface

Ansible is one of the most popular tools today in the Infrastructure Automation space. It is an IT orchestration engine that can be used in several subverticals, such as configuration management, orchestration, provisioning, and deployment. When compared to other automation tools, Ansible brings you an easy way to configure your infrastructure without the overhead of a client setup.

We started using Ansible to set up our build agents for Continuous Integration (CI) systems and were soon able to set up close to 150 agents that had different applications, within a couple of weeks. At that stage, we primarily used it for configuration management of our build agents, but once we had tasted success, we realized that it would help us solve our problems in production as well.
In production, we used another tool for configuration management for a long time, but we needed something else that would help us in orchestration and provisioning. Our deployments and orchestrations were very manual in nature and quite error-prone. We adopted Ansible in our production environments to solve this very problem at the start of the year, and we've had great success so far. We're able to build and rebuild infrastructures at a much faster rate that's less error-prone, involves less manual effort, and uses just a single click. Our deployment times have drastically come down. We've not yet attained Nirvana in this area but we're in a way better situation today compared to a year ago. We learned a lot in the process and we'd like to share our experiences with Ansible in this book. The Ansible website has been of great help along with the very intelligent community of Ansible.

We'd like to talk briefly about DevOps before moving ahead with Ansible. DevOps has been a movement aimed at bringing development and operations teams in organizations together a lot more than usual to release software earlier. Before the DevOps movement came into being, for a long time the notion was that development teams implement the features and throw the software over the wall for the operations team to then take it and run it in production. As a result, the operations teams had no idea what they were deploying, and the developers didn't know how their software was going to be deployed!

Another common complaint that most operations folks have heard is a developer coming to them and saying, "It works on my machine". An operations person recently responded to a similar question by saying, "Let's take your laptop and put it in production". Jokes apart, this isn't the ideal situation, is it? There has been a constant effort to get the development teams and operation teams to work a lot closer through a combination of tools and culture. Without the right culture, irrespective of what tools you use in your organization, the impact isn't going to be massive. However, changing and adopting the best practices between teams and breaking the wall, so to speak, between these diverse teams is what the DevOps movement advocates.

Several teams have begun to embed operations folks in their development workflows, right from the planning phase, so that the operations teams are aware of what's going on; at the same time, they can offer their expertise during the early stages of development. This is an aspect of culture that you can bring about in your organization and that would reduce the risk and help you release your software faster to production. For example, insights around scaling, database replication, and logging practices are skills that operations teams bring in. On the other hand, skills such as testing individual components and using unit, integration, and functional testing are what the operations teams can ideally pick up from the developers.

The other aspect of DevOps that we mentioned earlier was the tools. With the advent and adoption of cloud, which basically introduced the concept of "on-demand" and "pay-as-you-go" computing, tooling has become all the more important. There are primary focus areas, however, where tools have made significant progress. Let's broadly look at these areas:

- **Configuration Management**: Several tools have come up in this area. The aim of configuration management is to make sure your machines attain the intended state as described in the configuration in the fastest possible time and in the right manner so that they can play a particular role in your environment. For example, if you have to introduce a new web server during a traffic surge, how quickly can you do it, once you have a machine is what configuration management addresses. This also has resulted in the operations folks writing code and it's commonly termed as "Infrastructure as code", since all the code that is necessary to set up your infrastructure is now stored in the source control. This has slowly led to the adoption of Software Development Lifecycle (SDLC) for infrastructure code. This includes tools that aid your infrastructure testing. Tools in this space include CFEngine, Chef, Puppet, Ansible, Salt, and so on. Infrastructure-testing tools include Serverspec, Test kitchen, and so on.

- **Provisioning**: The tools in this space address how quickly you can bring up new machines in your data center, virtualized environment, or your cloud. Almost all cloud providers have APIs. The tools in this space use these APIs to speed up instance provisioning. For organizations that are on Linux, containers have made rapid strides in the last year or so with solutions such as Docker and LXCs in the forefront, and more and more people are beginning to use tools to make sure their containers are provisioned in an automated way. Ansible plays an important role in both these scenarios. Finally, there are tools such as Vagrant, which help you automate development and test environments.

- **Deployment**: Tools in this area focus on how you can deploy applications with zero downtime and in the best possible way. Many organizations now perform Rolling deployments or Canary deployments. Ansible supports both. Deployment pipelines have also become popular and tools such as ThoughtWorks Go, Atlassian Bamboo, and Jenkins with its innumerable plugins are strong players in this area.

- **Orchestration**: Tools in this area focus on how to coordinate among various components in your infrastructure so as to perform deployments. For example, making sure a web server is disabled from a load balancer, before releasing a new version of your software to the web server, is a common and famous example. Ansible, Mcollective, Salt, Serf, and Chef are some of the tools that help in this area.

- **Monitoring and Alerting**: Monitoring and alerting tools have evolved to handle fast-growing massive server environments. Legacy tools such as Nagios, Ganglia, and Zenoss along with newer tools such as Graphite, Sensu, and Riemann play a major role in this domain.

- **Logging**: Centralized logging makes sure you collect the right set of logs across systems and applications so that you can write rules on top of them to perform intelligent deductions, be it root cause analysis or alerting. Tools such as Logstash-Kibana, SumoLogic, and Rsyslog are quite popular in this space.

Ansible plays a significant role in four of the six major areas where tooling plays a very important role. Along with this, the people who can contribute heavily to these areas include sysadmins, operations teams, infrastructure admins, developers, and anyone with the mindset of bringing about infrastructure automation. The book aims to help and guide all these categories of users to set up robust automation for their infrastructure.

What this book covers

Chapter 1, *Getting Started with Ansible*, teaches you the basics of Ansible, its architecture, and how to set it up and get started. It starts with Ansible's "Hello, world!" program and builds the rest of the examples on top of it. You'll be introduced to inventories, modules, variables, and playbooks, and how Ansible can be used for configuration management.

Chapter 2, *Developing, Testing, and Releasing Playbooks*, will focus on how you can develop your Ansible playbooks, test them, how to handle multiple environments, and how to release your Ansible code into production. It also discusses the Software Development Life Cycle (SDLC), which is as important with an infrastructure management tool development as it is with any other custom software that is built.

Chapter 3, *Taking Ansible to Production*, focuses on all the important features that you would require for taking Ansible into production, more from a configuration management perspective. You will learn about features such as include, loops, and conditions in Ansible; handlers and security management with Ansible; and, most importantly, how to model your infrastructure using Roles. With the knowledge gained from the first three chapters, you will know enough to write playbooks that can be deployed in Production to configure your infrastructure.

Chapter 4, *Error Handling, Rollback, and Reporting,* helps you with topics such as how to debug, rollback, and report what you have in production. In almost all cases, you need to have enough information regarding these topics. It introduces Callback plugins and techniques you can use to rollback configurations when something goes wrong. It shows you how you can integrate Ansible with alerting and monitoring tools, and generate metrics for error handling and reporting. In short, you will be introduced to all that you would like in your sysadmin avatar.

Chapter 5, *Working with Custom Modules*, runs you through how you can write custom modules for your setups. One of the strengths of Ansible is that you can write custom modules in any language that has an interpreter (if the language is available on the box), provided that they return a JSON output. In most cases, we have seen that, having intelligent modules reduce the size of your playbooks and make them more readable.

Chapter 6, *Provisioning*, explains how you can bring up new instances in clouds as part of your provisioning activity. With the advent of cloud, the demand for spawning machines in clouds, such as AWS, Rackspace, and DigitalOcean, have gone up quite significantly. We'll also look at one of the most exciting container technologies, Docker, and how you can use Ansible to provision new Docker containers.

Chapter 7, *Deployment and Orchestration*, looks at how you can deploy Rails as well as Tomcat applications, along with packaging and deployment techniques. For a large infrastructure, it's important to deploy software in as little time as possible and, in almost all cases, with zero downtime; Ansible plays a key role in deployment and orchestration. We'll also look at how you can use the Ansible pull when you have large infrastructures.

Appendix, *Ansible on Windows, Ansible Galaxy, and Ansible Tower*, discusses Ansible's support for Windows. In addition, we'll cover Ansible Galaxy, a free site from where you can download community roles and get started, and finally, Ansible Tower, a web-based GUI developed by Ansible that provides you with a dashboard to manage your nodes via Ansible.

What you need for this book

You will need the following software to learn and execute the code files provided with this book:

- Ansible along with the required Python packages. This is covered in more detail in the book.
- Vagrant and serverspec to test code samples.

You will also need to install the following packages:

- pip
- Paramiko
- PyYAML
- Jinja2
- httplib2
- Git

Who this book is for

If you want to learn how to use Ansible to automate an infrastructure, either from scratch or to augment your current tooling with Ansible, then this book is for you. It has plenty of practical examples to help you get to grips with Ansible.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The Ansible template also has a `validate` parameter, which allows you to run a command to validate the file before copying it."

Any command-line input or output is written as follows:

```
$ git diff playbooks/example1.yml


   - name: Check httpd service
   - service: name=httpd state=started
   + service: name=httpd state=started enabled=yes
   - sudo: yes
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Vagrant defines a separate term called **Provisioning**, which runs the configuration management blocks or blocks that are plugged into the `Vagrantfile`, be it Ansible, Chef, Puppet, or shell scripts."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with Ansible

We keep moving forward, opening new doors, and doing new things, because we're curious and curiosity keeps leading us down new paths.

- Walt Disney

If exploring new paths is what you like, then in this chapter, we're going to lead you down an exciting path with **Ansible**. Almost always, when one starts to invest time trying to learn a new tool or language, the expectation is to install a new tool, get a "Hello, world!" example out of the way, tweet about it (these days), and continue learning more features in the general direction of solving the problem at hand. The aim of this chapter is to make sure all of this (and more) is achieved.

In this chapter, we will cover the following topics:

- What is Ansible?
- The Ansible architecture
- Configuring Ansible
- Configuration management
- Working with playbooks
- Variables and their types
- Working with inventory files
- Working with modules

At the end of this chapter, you will be able to create basic playbooks and understand how to use Ansible.

What is Ansible?

Ansible is an orchestration engine in IT, which can be used for several use cases. Compared to other automation tools, Ansible brings you an easy way to configure your orchestration engine without the overhead of a client or central server setup. That's right! No central server! It comes preloaded with a wide range of modules that make your life simpler.

In this chapter, you will learn the basics of Ansible and how to set up Ansible on your system. Ansible is an open source tool (with enterprise editions available) developed using Python and runs on Windows, Mac, and UNIX-like systems. You can use Ansible for configuration management, orchestration, provisioning, and deployments, which covers many of the problems that are solved under the broad umbrella of **DevOps**. We won't be talking about culture here as that's a book by itself!

> You could refer to the book, *Continuous Delivery and DevOps – A Quickstart Guide* by *Packt Publishing* for more information.

Let's try to answer some questions that you may have right away.

- *Can I use Ansible if I am starting afresh, have no automation in my system, and would like to introduce that (and as a result, increase my bonus for the next year)?*

    A short answer to this question is Ansible is probably perfect for you. The learning curve with Ansible is way shorter than most other tools currently present in the market. For the long answer, you need to read the rest of the chapters!

- *I have other tools in my environment. Can I still use Ansible?*

    Yes, again! If you already have other tools in your environment, you can still augment those with Ansible as it solves many problems in an elegant way. A case in point is a puppet shop that uses Ansible for orchestration and provisioning of new systems but continues to use Puppet for configuration management.

- *I don't have Python in my environment and introducing Ansible would bring in Python. What do I do?*

    You need to remember that, on most Linux systems, a version of Python is present at boot time and you don't have to explicitly install Python. You should still go ahead with Ansible if it solves particular problems for you. Always question what problems you are trying to solve and then check whether a tool such as Ansible would solve that use case.

- *I have no configuration management at present. Can I start today?*

  The answer is yes!

In many of the conferences we presented, the preceding four questions popped up most frequently. Now that these questions are answered, let's dig deeper.

The architecture of Ansible is agentless. Yes, you heard that right; you don't have to install any client-side software. It works purely on SSH connections; so, if you have a well oiled-SSH setup, then you're ready to roll Ansible into your environment in no time. This also means that you can install it only on one system (either a Linux or Mac machine) and you can control your entire infrastructure from that machine.

Yes, we understand that you must be thinking about what happens if this machine goes down. You would probably have multiple such machines in production, but this was just an example to elucidate the simplicity of Ansible. You could even run some of these machines from where you kick off Ansible scripts in a **Demilitarized Zone** (**DMZ**) to deal with your production machines.

The following table shows a small comparison of agentless versus agent-based configuration management systems:

Agent-based systems	Agentless systems
These systems need an agent and its dependencies installed.	No specific agent or third-party dependencies are installed on these systems. However, you need an SSH daemon that's up and running, in most cases.
These systems need to invoke the agent to run the configuration management tool. They can run it as a service or cron job. No external invocation is necessary.	These systems invoke the run remotely.
Parallel agent runs might be slow if they all hit the same server and the server cannot process several concurrent connections effectively. However, if they run without a server, the run would be faster.	Parallel agent runs might be faster than when all agents are contacting the same machine, but they might be constrained by the number of SSH connections since the runs are being invoked remotely.
The agent's installation and permissions need to be taken care of along with the configuration of the agent itself, for example, the server that they should talk to.	Remote connections can log in as a specific user and with the right level of user support since it's SSH-based.

Ansible primarily runs in the **push mode** but you can also run Ansible using `ansible-pull`, where you can install Ansible on each agent, download the playbooks locally, and run them on individual machines. If there is a large number of machines (large is a relative term; in our view, greater than 500 and requiring parallel updates) and you plan to deploy updates to the machine in parallel, this might be the right way to go about it.

To speedup default SSH connections, you can always enable `ControlPersist` and the pipeline mode, which makes Ansible faster and secure. Ansible works like any other UNIX command that doesn't require any daemon process to be running all the time.

Tools such as **Chef** and **Puppet** are agent-based and they need to communicate with a central server to pull updates. These can also run without a centralized server to scale a large number of machines commonly called **Serverless Chef** and **Masterless Puppet**, respectively.

When you start with something new, the first aspect you need to pay attention to is the nomenclature. The faster you're able to pick up the terms associated with the tool, the faster you're comfortable with that tool. So, to deploy, let's say, a package on one or more machines in Ansible, you would need to write a playbook that has a single task, which in turn uses the `package` module that would then go ahead and install the package based on an inventory file that contains a list of these machines. If you feel overwhelmed by the nomenclature, don't worry, you'll soon get used to it. Similar to the `package` module, Ansible comes loaded with more than 200 modules, purely written in Python. We will talk about modules in detail in the later chapters.

It is now time to install Ansible to start trying out various fun examples.

Installing Ansible

Installing Ansible is rather quick and simple. You can directly use the source code by cloning it from the GitHub project (`https://github.com/ansible/ansible`), install it using your system's package manager, or use Python's package management tool (**pip**). You can use Ansible on any Windows, Mac, or UNIX-like system. Ansible doesn't require any database and doesn't need any daemons running. This makes it easier to maintain the Ansible versions and upgrade without any breaks.

We'd like to call the machine where we will install Ansible our **command center**. Many people also refer to it as the **Ansible workstation**.

Note that, as Ansible is developed using Python, you would need Python Version 2.4 or a higher version installed. Python is preinstalled, as specified earlier, on the majority of operating systems. If this is not the case for you, refer to `https://wiki.python.org/moin/BeginnersGuide/Download` to download/upgrade Python.

Installing Ansible from source

Installing from source is as easy as cloning a repository. You don't require any root permissions while installing from source. Let's clone a repository and activate **virtualenv**, which is an isolated environment in Python where you can install packages without interfering with the system's Python packages. The command and the resultant output for the repository is as follows:

```
$ git clone git://github.com/ansible/ansible.git
Initialized empty Git repository in /home/vagrant/ansible/.git/
remote: Counting objects: 67818, done.
remote: Compressing objects: 100% (84/84), done.
remote: Total 67818 (delta 49), reused 2 (delta 0)
Receiving objects: 100% (67818/67818), 21.06 MiB | 238 KiB/s, done.
Resolving deltas: 100% (42682/42682), done.


[node ~]$ cd ansible/
[node ansible]$ source ./hacking/env-setup


Setting up Ansible to run out of checkout...


PATH=/home/vagrant/ansible/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/
sbin:/usr/sbin:/sbin:/home/vagrant/bin
PYTHONPATH=/home/vagrant/ansible/lib:
MANPATH=/home/vagrant/ansible/docs/man:


Remember, you may wish to specify your host file with -i


Done!
```

Ansible needs a couple of Python packages, which you can install using pip. If you don't have pip installed on your system, install it using the following command. If you don't have `easy_install` installed, you can install it using Python's `setuptools` package on Red Hat systems or using Brew on the Mac:

```
$ sudo easy_install pip
<A long output follows>
```

Once you have installed pip, install the `paramiko`, `PyYAML`, `jinja2,` and `httplib2` packages using the following command lines:

```
$ sudo pip install paramiko PyYAML jinja2 httplib2

Requirement already satisfied (use --upgrade to upgrade): paramiko in /
usr/lib/python2.6/site-packages

Requirement already satisfied (use --upgrade to upgrade): PyYAML in /usr/
lib64/python2.6/site-packages

Requirement already satisfied (use --upgrade to upgrade): jinja2 in /usr/
lib/python2.6/site-packages

Requirement already satisfied (use --upgrade to upgrade): httplib2 in /
usr/lib/python2.6/site-packages

Downloading/unpacking markupsafe (from jinja2)

  Downloading MarkupSafe-0.23.tar.gz

  Running setup.py (path:/tmp/pip_build_root/markupsafe/setup.py) egg_
info for package markupsafe

Installing collected packages: markupsafe

  Running setup.py install for markupsafe

    building 'markupsafe._speedups' extension

    gcc -pthread -fno-strict-aliasing -O2 -g -pipe -Wall -Wp,-D_FORTIFY_
SOURCE=2 -fexceptions -fstack-protector --param=ssp-buffer-size=4 -m64
-mtune=generic -D_GNU_SOURCE -fPIC -fwrapv -DNDEBUG -O2 -g -pipe -Wall
-Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstack-protector --param=ssp-
buffer-size=4 -m64 -mtune=generic -D_GNU_SOURCE -fPIC -fwrapv -fPIC
-I/usr/include/python2.6 -c markupsafe/_speedups.c -o build/temp.
linux-x86_64-2.6/markupsafe/_speedups.o

    gcc -pthread -shared build/temp.linux-x86_64-2.6/markupsafe/_
speedups.o -L/usr/lib64 -lpython2.6 -o build/lib.linux-x86_64-2.6/
markupsafe/_speedups.so

Successfully installed markupsafe

Cleaning up...
```

> By default, Ansible will be running against the development branch. You might want to check out the latest stable branch. Check what the latest stable version is using the following command line:
>
> **$ git branch -a**

Copy the latest version you want to use. Version 1.7.1 was the latest version available at the time of writing. Check the latest version you would like to use using the following command lines:

```
[node ansible]$ git checkout release1.7.1

Branch release1.7.1 set up to track remote branch release1.7.1 from
origin.

Switched to a new branch 'release1.7.1'


[node ansible]$ansible --version

ansible 1.7.1 (release1.7.1 268e72318f) last updated 2014/09/28 21:27:25
(GMT +000)
```

You now have a working setup of Ansible ready. One of the benefits of running Ansible through source is that you can enjoy the benefits of new features immediately without waiting for your package manager to make them available for you.

Installing Ansible using the system's package manager

Ansible also provides a way to install itself using the system's package manager. We will look into installing Ansible via **Yum**, **Apt**, **Homebrew**, and **pip**.

Installing via Yum

If you are running a Fedora system, you can install Ansible directly. For CentOS- or RHEL-based systems, you should add the EPEL repository first, as follows:

```
$ sudo yum install ansible
```

> On Cent 6 or RHEL 6, you have to run the command `rpm -Uvh`. Refer to `http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm` for instructions on how to install EPEL.

You can also install Ansible from an RPM file. You need to use the `make rpm` command against the `git` checkout of Ansible, as follows:

```
$ git clone git://github.com/ansible/ansible.git
$ cd ./ansible
$ make rpm
$ sudo rpm -Uvh ~/rpmbuild/ansible-*.noarch.rpm
```

> You should have `rpm-build`, `make`, and `python2-devel` installed on your system to build an rpm.

Installing via Apt

Ansible is available for Ubuntu in a **Personal Package Archive** (**PPA**). To configure the PPA and install Ansible on your Ubuntu system, use the following command lines:

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

You can also compile a deb file for Debian and Ubuntu systems, using the following command line:

```
$ make deb
```

Installing via Homebrew

You can install Ansible on Mac OSX using Homebrew, as follows:

```
$ brew update
$ brew install ansible
```

Installing via pip

You can install Ansible via Python's package manager pip. If you don't have pip installed on your system, install it. You can use pip to install Ansible on Windows too, using the following command line:

```
$ sudo easy_install pip
```

You can now install Ansible using pip, as follows:

```
$ sudo pip install ansible
```

Once you're done installing Ansible, run `ansible --version` to verify that it has been installed:

```
$ ansible –version
```

You will get the following as the output of the preceding command line:

```
ansible 1.7.1
```

Hello Ansible

Let's start by checking if two remote machines are reachable; in other words, let's start by pinging two machines following which we'll echo `hello ansible` on the two remote machines. The following are the steps that need to be performed:

1. Create an Ansible inventory file. This can contain one or more groups. Each group is defined within square brackets. This example has one group called `servers`:

    ```
    $ cat inventory
    [servers]
    machine1
    machine2
    ```

2. Now, we have to ping the two machines. In order to do that, first run `ansible --help` to view the available options, as shown below (only copying the subset that we need for this example):

    ```
    ansible --help
    Usage: ansible <host-pattern> [options]
    Options:
    -a MODULE_ARGS, --args=MODULE_ARGS
                            module arguments
    -i INVENTORY, --inventory-file=INVENTORY
                            specify inventory host file
                            (default=/etc/ansible/hosts)
    -m MODULE_NAME, --module-name=MODULE_NAME
                            module name to execute
                            (default=command)
    ```

We'll now ping the two servers using the Ansible command line, as shown in the following screenshot:

```
$ ansible servers -m ping -i inventory -u ec2-user
machine1 | success >> {
    "changed": false,
    "ping": "pong"
}

machine2 | success >> {
    "changed": false,
    "ping": "pong"
}
```

3. Now that we can ping these two servers, let's echo `hello ansible!`, using the command line shown in the following screenshot:

```
$ ansible servers -m shell -a '/bin/echo hello ansible!' -i inventory -u ec2-user
machine2 | success | rc=0 >>
hello ansible!

machine1 | success | rc=0 >>
hello ansible!
```

Consider the following command:

`$ansible servers -i inventory -a '/bin/echo hello ansible!'`

The preceding command line is the same as the following one:

`$ansible servers -i inventory  -m command -a '/bin/echo hello ansible!'.`

If you move the inventory file to /etc/ansible/hosts, the Ansible command will become even simpler, as follows:

`$ ansible servers -a '/bin/echo hello ansible!'`

There you go. The 'Hello Ansible' program works! Time to tweet!
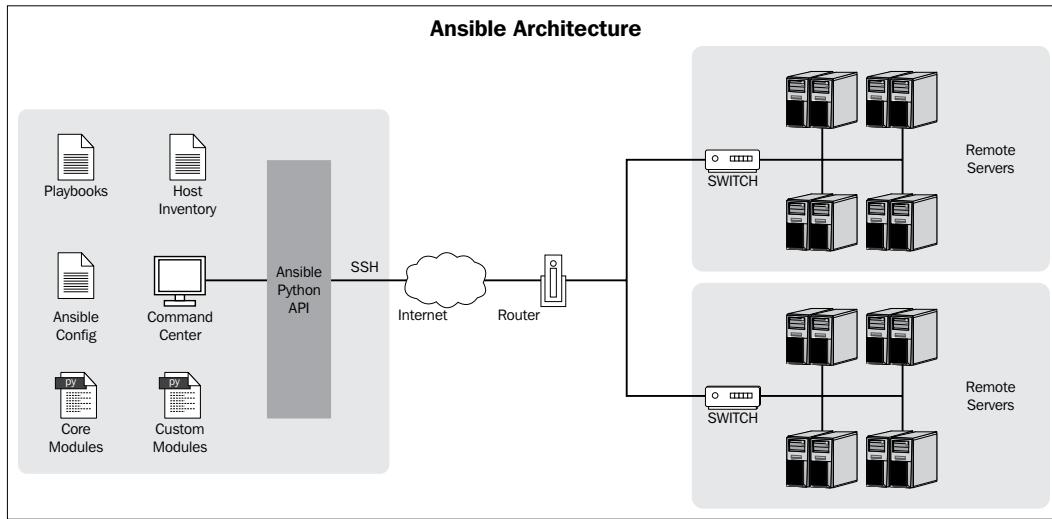
> You can also specify the inventory file by exporting it in a variable named ANSIBLE_HOSTS. The preceding command, without the -i option, will work even in that situation.

Now that we've seen the 'Hello, world!' example, let's dig a little deeper into the architecture. Once you've had a hand on the architecture, you will start realizing the power of Ansible.

The Ansible architecture

As you can see from the following diagram, the idea is to have one or more command centers from where you can blast out commands onto remote machines or run a sequenced instruction set via playbooks:

The host inventory file determines the target machines where these **plays** will be executed. The Ansible configuration file can be customized to reflect the settings in your environment. The remote servers should have Python installed along with a library named `simplejson` in case you are using Python Version 2.5 or an earlier version.

The playbooks consist of one or more tasks that are expressed either with core modules that come with Ansible or custom modules that you can write for specific situations. The plays are executed sequentially from top to bottom, so there is no explicit order that you have to define. However, you can perform conditional execution on tasks so that they can be **skipped** (an Ansible term) if the conditions are not met.

You can also use the Ansible API to run scripts. These are situations where you would have a wrapper script that would then utilize the API to run the playbooks as needed. The playbooks are declarative in nature and are written in **YAML Ain't Markup Language** (**YAML**). This takes the declarative simplicity of such systems to a different level.

Ansible can also be used to provision new machines in data centers and/or Cloud, based on your infrastructure and configure them based on the **role** of the new machine. For such situations, Ansible has the power to execute a certain number of tasks in the **local** mode, that is, on the command center, and certain tasks on the actual machine, post the machine-boot-up phase.

In this case, a local action can spawn a new machine using an API of some sort, wait for the machine to come up by checking whether standard ports are up, and then log in to the machine and execute commands. The other aspect to consider is that Ansible can run tasks either serially or *N* threads in parallel. This leads to different permutations and combinations when you're using Ansible for deployment.

Before we proceed with full-fledged examples and look at the power of Ansible, we'll briefly look at the Ansible configuration file. This will let you map out the configuration to your setup.

Configuring Ansible

An Ansible configuration file uses an INI format to store its configuration data. In Ansible, you can overwrite nearly all of the configuration settings either through Ansible playbook options or environment variables. While running an Ansible command, the command looks for its configuration file in a predefined order, as follows:

1. `ANSIBLE_CONFIG`: Firstly, the Ansible command checks the environment variable, which points to the configuration file
2. `./ansible.cfg`: Secondly, it checks the file in the current directory
3. `~/.ansible.cfg`: Thirdly, it checks the file in the user's home directory
4. `/etc/ansible/ansible.cfg`: Lastly, it checks the file that is automatically generated when installing Ansible via a package manager

If you have installed Ansible through your system's package manager or pip, then you should already have a copy of `ansible.cfg` under the `/etc/ansible` directory. If you installed Ansible through the GitHub repository, you can find `ansible.cfg` under the `examples` directory, where you cloned your Ansible repository.

Configuration using environment variables

You can use most of the configuration parameters directly via environment variables by appending `ANSIBLE_` to the configuration parameter (the parameter name should be in uppercase). Consider the following command line for example:

export ANSIBLE_SUDO_USER=root

The `ANSIBLE_SUDO_USER` variable can then be used as part of the playbooks.

Configuration using ansible.cfg

Ansible has many configuration parameters; you might not need to use all of them. We will consider some of the configuration parameters, as follows, and see how to use them:

- `hostfile`: This parameter indicates the path to the inventory file. The inventory file consists of a list of hosts that Ansible can connect to. We will discuss inventory files in detail later in this chapter. Consider the following command line for example:

  hostfile = /etc/ansible/hosts

- `library`: Whenever you ask Ansible to perform any action, whether it is a local action or a remote one, it uses a piece of code to perform the action; this piece of code is called a module. The `library` parameter points to the path of the directory where Ansible modules are stored. Consider the following command line for example:

  library = /usr/share/ansible

- `forks`: This parameter is the default number of processes that you want Ansible to spawn. It defaults to five maximum processes in parallel. Consider the following command line for example:

  forks = 5

- `sudo_user`: This parameter specifies the default user that should be used against the issued commands. You can override this parameter from the Ansible playbook as well (this is covered in a later chapter). Consider the following command line for example:

  sudo_user = root

- `remote_port`: This parameter is used to specify the port used for SSH connections, which defaults to `22`. You might never need to change this value unless you are using SSH on a different port. Consider the following command line for example:

  remote_port = 22

- `host_key_checking`: This parameter is used to disable the SSH host key checking; this is set to `True` by default. Consider the following command line for example:

  `host_key_checking = False`

- `timeout`: This is the default value for the timeout of SSH connection attempts:

  `timeout = 60`

- `log_path`: By default, Ansible doesn't log anything; if you would like to send the Ansible output to a logfile, then set the value of `log_path` to the file you would like to store the Ansible logs in. Consider the following command line for example:

  `log_path = /var/log/ansible.log`

In the latter half of this chapter, we'll focus on Ansible features and, primarily, how it can be used for configuration management. We'd recommend you to try out the given examples.

Configuration management

There has been a huge shift across companies of all sizes throughout the world in the field of infrastructure automation. **CFEngine** was one of the first tools to demonstrate this capability way back in the 1990s; more recently, there have been Puppet, Chef, and Salt besides Ansible. We will try and compare Ansible with Puppet and Chef during the course of this book since we've had a good experience with all three tools. We will also point out specifically how Ansible would solve a problem compared to Chef or Puppet.

All of them are declarative in nature and expect to move a machine to the desired state that is specified in the configuration. For example, in each of these tools, in order to start a service at a point in time and start it automatically on restart, you would need to write a declarative block or module; every time the tool runs on the machine, it will aspire to obtain the state defined in your playbook (Ansible), cookbook (Chef), or manifest (Puppet).

The difference in the toolset is minimal at a simple level but as more situations arise and the complexity increases, you will start finding differences between the different toolsets. In Puppet, you need to take care of the order, and the puppet server will create the sequence of instructions to execute every time you run it on a different box. To exploit the power of Chef, you will need a good Ruby team. Your team needs to be good at the Ruby language to customize both Puppet and Chef, and you will need a bigger learning curve with both the tools.

With Ansible, the case is different. It uses the simplicity of Chef when it comes to the order of execution, the top-to-bottom approach, and allows you to define the end state in the YAML format, which makes the code extremely readable and easy for everyone, from Development teams to Operations teams, to pick up and make changes. In many cases, even without Ansible, operations teams are given playbook manuals to execute instructions from, whenever they face issues. Ansible mimics that behavior. Do not be surprised if you end up having your project manager change the code in Ansible and check it into git because of its simplicity!

Let's now start looking at playbooks, variables, inventory files, and modules.

Working with playbooks

Playbooks are one of the core features of Ansible and tell Ansible what to execute. They are like a to-do list for Ansible that contains a list of tasks; each task internally links to a piece of code called a module. Playbooks are simple human-readable YAML files, whereas modules are a piece of code that can be written in any language with the condition that its output should be in the JSON format. You can have multiple tasks listed in a playbook and these tasks would be executed serially by Ansible. You can think of playbooks as an equivalent of manifests in Puppet, states in Salt, or cookbooks in Chef; they allow you to enter a list of tasks or commands you want to execute on your remote system.

The anatomy of a playbook

Playbooks can have a list of remote hosts, user variables, tasks, handlers (covered later in this chapter), and so on. You can also override most of the configuration settings through a playbook. Let's start looking at the anatomy of a playbook. The purpose of a playbook is to ensure that the `httpd` package is installed and the service is started. Consider the following screenshot, where the `setup_apache.yml` file is shown:

```
$ cat playbooks/setup_apache.yml
---
-   hosts: host1
    remote_user: ec2-user
    tasks:

      - name: Install httpd package
        yum: name=httpd state=latest
        sudo: yes

      - name: Starting httpd service
        service: name=httpd state=started
        sudo: yes
```

The `setup_apache.yml` file is an example of a playbook. The file comprises of three main parts, as follows:

- `hosts`: This lists the host or host group against which we want to run the task. The `hosts` field is mandatory and every playbook should have it (except roles). It tells Ansible where to run the listed tasks. When provided with a host group, Ansible will take the host group from the playbook and will try looking for it in an inventory file (covered later in the chapter). If there is no match, Ansible will skip all the tasks for that host group. The `--list-hosts` option along with the playbook (`ansible-playbook <playbook> --list-host`) will exactly tell you against which hosts the playbook will run.

- `remote_user`: This is one of the configuration parameters of Ansible (consider, for example, `tom' - remote_user`) that tells Ansible to use a particular user (in this case, `tom`) while logging into the system.

- `tasks`: Finally, we come to tasks. All playbooks should contain tasks. Tasks are a list of actions you want to perform. A `tasks` field contains the name of the task, that is, the help text for the user about the task, a module that should be executed, and arguments that are required for the module. Let's look at the single task that is listed in the playbook, as shown in the preceding screenshot:

```
tasks:
  - name: Install httpd package
    yum: name=httpd state=latest
    sudo: yes

  - name: Starting httpd service
    service: name=httpd state=started
    sudo: yes
```

> Most of the examples in the book would be executed on CentOS, but the same set of examples with a few changes would work on Ubuntu as well.

In the preceding case, there are two tasks. The `name` parameter represents what the task is doing and is present only to improve readability, as we'll see during the playbook run. The `name` parameter is optional. The modules, `yum` and `service`, have their own set of parameters. Almost all modules have the `name` parameter (there are exceptions such as the `debug` module), which indicates what component the actions are performed on. Let's look at the other parameters:

- In the `yum` module's case, the `state` parameter has the `latest` value and it indicates that the `httpd latest` package should be installed. The command to execute more or less translates to `yum install httpd`.

- In the `service` module's scenario, the `state` parameter with the `started` value indicates that `httpd service` should be started, and it roughly translates to `/etc/init.d/httpd start`.

- The `sudo: yes` parameter represents the fact that the tasks should be executed with the `sudo` access. If the `sudo` user's file does not allow the user to run the particular command, then the playbook will fail when it is run.

> You might have questions about why there is no `package` module that internally figures out the architecture and runs either the `yum`, `apt`, or other package options depending on the architecture of the system. Ansible populates the package manager value into a variable named `ansible_pkg_manager`.
>
> In general, we need to remember that the number of packages that have a common name across different operating systems is a small subset of the number of packages that are actually present. For example, the `httpd` package is called `httpd` in Red Hat systems and `apache2` in Debian-based systems. We also need to remember that every package manager has its own set of options that make it powerful; as a result, it makes more sense to use explicit package manager names so that the full set of options are available to the end user writing the playbook.

Let's look at the folder structure before we run the playbook. We have a folder named `example1`; within that, there are different files related to the Ansible playbook. With advanced examples, we'll see various folders and multiple files. Consider the following screenshot:

```
$ tree
.
├── hosts
└── playbooks
    └── setup_apache.yml

1 directory, 2 files
```

The `hosts` file is set up locally with one host named `host1` that corresponds to what is specified in the `setup_apache.yml` playbook:

```
[root@node example1]# cat hosts
host1
```

Now, it's time (yes, finally!) to run the playbook. Run the command line, as shown in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/setup_apache.yml

PLAY [host1] ******************************************************************

GATHERING FACTS **************************************************************
ok: [host1]

TASK: [Install httpd package] ************************************************
changed: [host1]

TASK: [Starting httpd service] ***********************************************
changed: [host1]

PLAY RECAP *******************************************************************
host1                      : ok=3    changed=2    unreachable=0    failed=0
```

Wow! The example worked. Let's now check whether the `httpd` package is installed and up-and-running on the machine. Perform the steps shown in the following screenshot:

```
[root@node ~]# rpm -qa | grep httpd
httpd-tools-2.2.27-1.2.amzn1.x86_64
httpd-2.2.27-1.2.amzn1.x86_64
[root@node ~]#
[root@node ~]# service httpd status
httpd (pid  2168) is running...
```

The end state, according to the playbook, has been achieved. Let's briefly look at what exactly happens during the playbook run:

#ansible-playbook -i hosts playbooks/setup_apache.yml

The command, `ansible-playbook`, is what you would use in order to invoke the process that runs a playbook. The familiar `-i` option points to the inventory host file. We'll look at other options with `ansible-playbook` in a later section.

Next, we'll look into the `Gathering Facts` task that, when run, is displayed as follows:

GATHERING FACTS *
****

ok: [host1]

The first default task when any playbook is run is the `Gathering Facts` task. The aim of this task is to gather useful metadata about the machine in the form of variables; these variables can then be used as a part of tasks that follow in the playbook. Examples of facts include the IP Address of the machine, the architecture of the system, and hostname. The following command will show you the facts collected by Ansible about the host:

```
ansible -m setup host1 -i hosts
```

You can disable the gathering of facts by setting the `gather_facts` command just below the `hosts` command in the Ansible playbook. We'll look at the pros and cons of fact gathering in a later chapter.

```
---
-   hosts: host1
    gather_facts: False
```

```
TASK: [Install httpd package] ****************************************
******
changed: [host1]
```

```
TASK: [Starting httpd service] ****************************************
******
changed: [host1]
```

Followed by the execution of the preceding command lines, we have the actual tasks that are executed. Both the tasks give their outputs stating whether the state of the machine has changed by running the task or not. In this case, since neither the `httpd` package was present nor the service was started, the tasks' outputs changed for the user to see on the screen (as seen in the preceding screenshot). Let's rerun the task now and see the output after both the tasks have actually run.

```
$ ansible-playbook -i hosts playbooks/setup_apache.yml

PLAY [host1] ********************************************************

GATHERING FACTS ****************************************************
ok: [host1]

TASK: [Install httpd package] *************************************
ok: [host1]

TASK: [Starting httpd service] ***********************************
ok: [host1]

PLAY RECAP ********************************************************
host1                      : ok=3    changed=0    unreachable=0    failed=0
```

As you would have expected, the two tasks in question give an output of `ok`, which would mean that the desired state was already met prior to running the task. It's important to remember that many tasks such as the `Gathering facts` task obtain information regarding a particular component of the system and do not necessarily change anything on the system; hence, these tasks didn't display the `changed` output earlier.

The `PLAY RECAP` section in the first and second run are shown as follows. You will see the following output during the first run:

```
PLAY RECAP ***********************************************************************
host1                      : ok=3    changed=2    unreachable=0    failed=0
```

You will see the following output during the second run:

```
PLAY RECAP ***********************************************************************
host1                      : ok=3    changed=0    unreachable=0    failed=0
```

As you can see, the difference is that the first task's output shows `changed=2`, which means that the system state changed twice due to two tasks. It's very useful to look at this output, since, if a system has achieved its desired state and then you run the playbook on it, the expected output should be `changed=0`.

If you're thinking of the word **Idempotency** at this stage, you're absolutely right and deserve a pat on the back! Idempotency is one of the key tenets of Configuration Management. Wikipedia defines Idempotency as an operation that, if applied twice to any value, gives the same result as if it were applied once. Earliest examples that you would have encountered in your childhood would be multiplicative operations on the number 1, where `1*1=1` every single time.

Most of the configuration management tools have taken this principle and applied it to the infrastructure as well. In a large infrastructure, it is highly recommended to monitor or track the number of changed tasks in your infrastructure and alert the concerned tasks if you find oddities; this applies to any configuration management tool in general. In an ideal state, the only time you should see changes is when you're introducing a new change in the form of any **Create, Remove, Update, or Delete** (**CRUD**) operation on various system components. If you're thinking how you can do it with Ansible, keep reading the book and you'll eventually find the answer!

Let's proceed. You could have also written the preceding tasks as follows but when the tasks are run, from an end user's perspective, they are quite readable:

```
tasks:
  - yum: name=httpd state=latest
    sudo: yes
  - service: name=httpd state=started
    sudo: yes
```

Let's run the playbook again to spot any difference in the output, as shown in the following screenshot:

As you can see, the difference is in the readability. Wherever possible, it's recommended to keep the tasks as simple as possible (the KISS principle of Keep It Simple Stupid) to allow for maintainability of your scripts in the long run.

Now that we've seen how you can write a basic playbook and run it against a host, let's look at other options that would help you while running playbooks.

One of the first options anyone picks up is the debug option. To understand what is happening when you run the playbook, you can run it with the **Verbose** (-v) option. Every extra v will provide the end user with more debug output. Let's see an example of using the playbook debug for a single task using the following debug options.

- The -v option provides the default output, as shown in the preceding screenshot.
- The -vv option adds a little more information, as shown in the following screenshot:

- The -vvv option adds a lot more information, as shown in the following screenshot. This shows the SSH command Ansible uses to create a temporary file on the remote host and run the script remotely.

```
TASK: [yum name=httpd state=latest] ****************************************
<host1> ESTABLISH CONNECTION FOR USER: ec2-user
<host1> REMOTE_MODULE yum name=httpd state=latest
<host1> EXEC ['ssh', '-C', '-tt', '-vvv', '-o', 'ControlMaster=auto', '-o', 'ControlPersist=60s', '-o', 'ControlPath=/
Users/ramesh/.ansible/cp/ansible-ssh-%h-%p-%r', '-o', 'KbdInteractiveAuthentication=no', '-o', 'PreferredAuthenticatio
ns=gssapi-with-mic,gssapi-keyex,hostbased,publickey', '-o', 'PasswordAuthentication=no', '-o', 'User=ec2-user', '-o',
'ConnectTimeout=10', 'host1', "/bin/sh -c 'mkdir -p $HOME/.ansible/tmp/ansible-tmp-1405444665.37-184092656716318 && ch
mod a+rx $HOME/.ansible/tmp/ansible-tmp-1405444665.37-184092656716318 && echo $HOME/.ansible/tmp/ansible-tmp-140544466
5.37-184092656716318'"]
<host1> PUT /var/folders/zf/thjwr8ts1_l_5g3px98_f3l40000gr/T/tmpteQxC8 TO /home/ec2-user/.ansible/tmp/ansible-tmp-1405
444665.37-184092656716318/yum
<host1> EXEC ['ssh', '-C', '-tt', '-vvv', '-o', 'ControlMaster=auto', '-o', 'ControlPersist=60s', '-o', 'ControlPath=/
Users/ramesh/.ansible/cp/ansible-ssh-%h-%p-%r', '-o', 'KbdInteractiveAuthentication=no', '-o', 'PreferredAuthenticatio
ns=gssapi-with-mic,gssapi-keyex,hostbased,publickey', '-o', 'PasswordAuthentication=no', '-o', 'User=ec2-user', '-o',
'ConnectTimeout=10', 'host1', u"/bin/sh -c 'LC_CTYPE=en_US.UTF-8 LANG=en_US.UTF-8 /usr/bin/python -tt /home/ec2-user/.
ansible/tmp/ansible-tmp-1405444665.37-184092656716318/yum; rm -rf /home/ec2-user/.ansible/tmp/ansible-tmp-1405444665.3
7-184092656716318/ >/dev/null 2>&1'"]
ok: [host1] => {"changed": false, "msg": "", "rc": 0, "results": ["All packages providing httpd are up to date"]}
```

From a playbook, it becomes important to view what the values of certain variables are. In order to help you, there is a helpful debug module that you can add to your `setup_apache.yml` playbook, as shown in the following screenshot:

```
- name: Show how debug works
  debug: msg={{ ansible_distribution }}
```

Let's run the `setup_apache.yml` playbook and see how the debug module works in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/setup_apache.yml

PLAY [host1] ************************************************************

GATHERING FACTS ********************************************************
ok: [host1]

TASK: [Install httpd package] *****************************************
ok: [host1]

TASK: [Starting httpd service] ***************************************
ok: [host1]

TASK: [Show how debug works] *****************************************
ok: [host1] => {
    "msg": "Amazon"
}

PLAY RECAP ***********************************************************
host1                      : ok=4    changed=0    unreachable=0    failed=0
```

This is also the first usage of the metadata that we've gathered from the machine. Here, we're outputting the value that is assigned to the ansible_distribution variable. Every Ansible variable that has the metadata starts with ansible_. The debug option can be used generously to help you in your overall usage of tasks. Also, as expected, there is no change in the state of the machine; hence, changed=0.

The next useful option with ansible-playbook is to simply list all the tasks that will be executed when you run a playbook. When you have several tasks and multiple playbooks that run as part of a playbook, this option would help you analyze a playbook when it is run. Let's look at an example in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/setup_apache.yml --list-tasks

playbook: playbooks/setup_apache.yml

  play #1 (host1):
    Install httpd package
    Starting httpd service
    Show how debug works
```

You also have the start-at option. It will start executing the task you specify. Let's look at an example in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/setup_apache.yml --start-at-task='Starting httpd service'

PLAY [host1] ********************************************************************

GATHERING FACTS ****************************************************************
ok: [host1]

TASK: [Starting httpd service] ************************************************
ok: [host1]

TASK: [Show how debug works] **************************************************
ok: [host1] => {
    "msg": "Amazon"
}

PLAY RECAP ********************************************************************
host1                      : ok=3    changed=0    unreachable=0    failed=0
```

As you can see, there is no `Install httpd package` task in the preceding screenshot, since it was skipped. Another related option that we should look at is the `step` option. With this, you can prompt the user to execute a task (or not). Let's look at an example in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/setup_apache.yml --step

PLAY [host1] ********************************************************

GATHERING FACTS ****************************************************
ok: [host1]
Perform task: Install httpd package (y/n/c): y

Perform task: Install httpd package (y/n/c):  ********************************
ok: [host1]
Perform task: Starting httpd service (y/n/c): n
Perform task: Show how debug works (y/n/c): y

Perform task: Show how debug works (y/n/c):  ********************************
ok: [host1] => {
    "msg": "Amazon"
}

PLAY RECAP *********************************************************
host1                      : ok=3    changed=0    unreachable=0    failed=0
```

In the preceding example, we didn't execute the `Starting httpd service` task. There are several more useful options that we will cover later in this chapter and in *Chapter 2, Developing, Testing, and Releasing Playbooks*, along with relevant examples. For now, let's jump into variables.

Variables and their types

Variables are used to store values that can be later used in your playbook. They can be set and overridden in multiple ways. Facts of machines can also be fetched as variables and used. Ansible allows you to set your variables in many different ways, that is, either by passing a variable file, declaring it in a playbook, passing it to the `ansible-playbook` command using `-e / --extra-vars`, or by declaring it in an inventory file (discussed later in this chapter).

Before we look at the preceding ways in a little more detail, let's look at some of the ways in which variables in Ansible can help you, as follows:

- Use them to specify the package name when you have different operating systems running, as follows:

  ```
  - set_fact package_name=httpd
  ```

```
    when: ansible_os_family == "Redhat"
- set_fact package_name=apache2
    when: ansible_os_family == "Debian"
```

The preceding task will set a variable `package_name` either with `httpd` or `apache2` depending on the OS family of your machine. We'll look at other facts that are fetched as variables shortly.

- Use them to store user values using a prompt in your Ansible playbook:

```
- name: Package to install
  pause: prompt="Provide the package name which you want to
install "
  register: package_name
```

The preceding task will prompt the user to provide the package name. The user input would then be stored in a variable named `package_name`.

- Store a list of values and loop it.
- Reduce redundancy if the same variables are called in multiple playbooks that refer to the same variable name. This is so that you can change the value in just one place and it gets reflected in every invocation of the variable.

The types of variables that Ansible supports are `String`, `Numbers`, `Float`, `List`, `Dictionary`, and `Boolean`.

Variable names

All variable names in Ansible should start with a letter. The name can have letters, numbers, and an underscore.

Valid variable names in Ansible

The following are a few examples of valid variable names in Ansible:

- `package_name`
- `package_name2`
- `user_input_package`
- `Package`

Invalid variable names in Ansible

The following are a few examples of invalid variable names in Ansible:

- `mysql version` (multiple words)
- `mysql.port` (a dot)
- `5` (a number)
- `user-input` (a hyphen)

You can define variables in Ansible at different hierarchy levels; let's see what those levels are and how you can override variables in that hierarchy.

Variables in an included task file

Variables in an included task file will override any other variables defined at different levels of hierarchy except the extra variables passed through the command line. We will see how command-line variables work later in this chapter. This override feature allows you to change the value of a variable during different tasks, making it more dynamic. This is one of the widely used variable features, where you want to assign a default value to the variable and override it during different tasks. Let's see an example of how this works in the following screenshot:

```
$ cat playbooks/install_apache.yml
  - set_fact: package_name=httpd
    when: ansible_os_family == "Redhat"

  - set_fact: package_name=apache2
    when: ansible_os_family == "Debian"

  - name: Install httpd package
    yum: name=httpd state=latest
    sudo: yes
    when: ansible_os_family == "Redhat"

  - name: Install apache2 package
    yum: name=apache2 state=latest
    sudo: yes
    when: ansible_os_family == "Debian"

$
$ cat playbooks/apache.yml
---
- hosts: host1
  remote_user: ec2-user
  tasks:
    - include: install_apache.yml

    - name: Check apache service
      service: name={{ package_name }} state=started
      sudo: yes
```

In the preceding example, we created two playbooks. One will set a fact for the package name and install Apache depending on the OS distribution; the second one will actually be executed by Ansible, which will first call `install_apache.yml` and make sure the Apache service is running on the remote host. To fetch the package name, we will directly use the `package_name` variable that was set by the `install_apache.yml` playbook.

Variables in a playbook

Variables in a playbook are set by the `vars:` key. This key takes a key-value pair, where the key is the variable name and the value is the actual value of the variable. This variable will overwrite other variables that are set by a global variable file or from an inventory file. We will now see how to set a variable in a playbook using the preceding example. This is demonstrated in the following screenshot:

```
$ cat playbooks/apache.yml
---
- hosts: host1
  remote_user: ec2-user

  vars:
    - package_name: "httpd"

  tasks:
    - include: install_apache.yml

    - name: Check apache service
      service: name={{ package_name }} state=started
      sudo: yes
```

The preceding example will first set the `package_name` variable with a default value of `httpd`; this value will be further overridden by a task, `install_apache.yml`, that we included. You can define multiple variables, each in a separate line.

Variables in a global file

Variables in Ansible can also be defined in a separate file; this allows you to separate your data (that is, variables) from your playbook. You can define as many variable files as you want; you just need to tell your playbook the files it needs to look at for variables. The format to define variables in a file is similar to the format of playbook variables. You provide the variable name and its value in a key-value pair and it follows a YAML format. Let's see how it works in the following screenshot:

```
$ cat /etc/ansible/common/vars/global.yml
---
package_name: "httpd"
cassandra_path: /opt/apache-cassandra

# Environment variables
proxy_env:
 AWS_ACCESS_KEY: "{{ lookup('env','ACCESS_KEY_ID') }}"
 AWS_SECRET_KEY: "{{ lookup('env','SECRET_ACCESS_KEY') }}"

# Default mount point
mount_point: "/dev/sdf"
```

The preceding example defines some variables where we directly pass a default value, whereas, for AWS_ACCESS_KEY and AWS_SECRET_KEY, we fetch the value from an environment variable using the lookup plugin of the Jinja templating language (more on this in later chapters). Anything that succeeds hash (#) is not interpreted by Ansible and is counted as a comment. You can also have comments after a variable is defined. For example, consider the following command line:

```
mount_point: "/dev/sdf"     # Default mount point
```

You tell Ansible which variable files need to be checked by using the vars_files key. You can specify multiple variable files in a playbook. Ansible will check for a variable in a bottom-to-top manner. Let's see how this works, in the following screenshot:

```
$ cat playbooks/apache.yml
---
- hosts: host1
  remote_user: ec2-user
  vars_files:
     - var1.yml
     - var3.yml
     - var2.yml

  tasks:
     - name: Check apache service
       service: name={{ package_name }} state=started
       sudo: yes
```

In the preceding example, Ansible will first check for a variable named `package_name` in the `var2.yml` file. It will stop further lookup if it finds the variable in `var2.yml`; if not, it will try searching for the variable in `var3.yml`, and other files if there are any more.

Facts as variables

You've already seen an example of how to use a fact, such as `ansible_distribution`, that is obtained as a fact from a machine. Let's look at a bigger list that you can access when you run `gather_facts`. The same set of facts can be seen by running the `setup` module without using the `ansible-playbook` command, and by using the `ansible` command as shown in the following command lines:

```
$ ansible 192.168.33.10 -i inventory -m setup
192.168.33.10 | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.2.15",
            "192.168.33.10"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fec9:399e",
            "fe80::a00:27ff:fe72:5c55"
        ],
        "ansible_architecture": "x86_64",
----
        "ansible_distribution_major_version": "6",
        "ansible_distribution_release": "Final",
        "ansible_distribution_version": "6.4",
        "ansible_domain": "localdomain"
----
        "ansible_swapfree_mb": 2559,
        "ansible_swaptotal_mb": 2559,
        "ansible_system": "Linux",
        "ansible_system_vendor": "innotek GmbH",
        "ansible_user_id": "vagrant",
        "ansible_userspace_architecture": "x86_64",
        "ansible_userspace_bits": "64",
```

These facts are now exposed as variables and can be used as part of playbooks. You will find more examples regarding this later in this book.

Command-line variables

Command-line variables are a great way to overwrite file/playbook variables. This feature allows you to give your user the power to pass the value of a variable directly from an `ansible-playbook` command. You can use the `-e` or `--extra-vars` options of the `ansible-playbook` command by passing a string of key-value pairs, separated by a whitespace. Consider the following command line:

```
ansible-playbook -i hosts --private-key=~/.ssh/ansible_key playbooks/
apache.yml --extra-vars "package_name=apache2"
```

The preceding `ansible-playbook` command will overwrite the `package_name` variable if it is mentioned in any of the variable files. Command-line variables will not override the variables that are set by the `set_fact` module. To prevent this type of overriding, we can use Ansible's conditional statements, which will override a variable only if it is not defined. We will discuss more about conditional statements in later chapters.

One of the commonly used command-line variables is `hosts`. Till now, we saw some Ansible playbook examples where we directly passed the hostname to the playbook. Instead of passing the hostname directly, you can use an Ansible variable and leave it to the end user to provide the hostname. Let's see how this works in the following screenshot:

```
$ cat playbooks/apache.yml
---
- hosts: "{{ nodes }}"
  remote_user: ec2-user
  vars_files:
    - var1.yml
    - var3.yml
    - var2.yml

  tasks:
    - name: Check apache service
      service: name={{ package_name }} state=started
      sudo: yes
```

In the preceding playbook, instead of directly using the hostname, we will now pass a variable named `nodes`. The `ansible-playbook` command for such a playbook will look as follows:

```
ansible-playbook -i hosts --private-key=~/.ssh/ansible_key playbooks/
apache.yml --extra-vars "nodes=host1"
```

The preceding `ansible-playbook` command will set the value of `nodes` as `host1`. The Ansible playbook will now use the value of the `nodes` variable against `hosts`. You can also pass a group of hosts by passing the hostname to the `nodes` variable (more on grouping hosts will be seen in the next section).

The last thing we'd like to cover in this section is typing out all the options every single time. You can export all of the options as environment variables, as shown in the following command lines, so that you don't have to type them all.

```
$ env | grep ANSIBLE
ANSIBLE_INVENTORY=inventory
ANSIBLE_HOSTS=hosts
ANSIBLE_SSH_PRIVATE_KEY=~/.ssh/ansible_key
```

Once you type the preceding command lines, the playbook command would resemble the following command line:

```
ansible-playbook playbooks/apache.yml --extra-vars "nodes=host1"
```

Variables in an inventory file

All of the preceding variables are applied globally to all hosts against which you are running your Ansible playbook. You might sometimes need to use a specific list of variables for a specific host. Ansible supports this by declaring your variables inside an inventory file. There are different ways to declare variables inside an inventory file; we will look at how an inventory file works and how to use Ansible with it in the next section of this chapter.
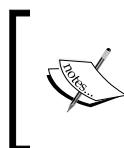
Apart from the user-defined variables, Ansible also provides some system-related variables called **facts**. These facts are available to all hosts and tasks, and are collected every time you run the `ansible-playbook` command, unless disabled manually. You can directly access these facts by using a Jinja template, for example, as follows:

```
- name: Show how debug works
  debug: msg={{ ansible_distribution }}
```

The `ansible_distribution` part in the preceding command line is a fact, which will be initialized by Ansible when you run the `ansible-playbook` command. To check all the facts Ansible collects, run the following command line:

```
ansible -m setup host1 -i host1,
```

The preceding example will run the setup module on `host1` and list out all possible facts that Ansible can collect from the remote host. It will also collect the facts from **facter** (a discovery program) if you have it installed on your system. The variable `-i` in the preceding example specifies an inventory file; in this case, instead of passing a file, we will directly use the hostname.

> When using a hostname directly instead of an inventory file, you need to add a trailing comma `","` with the hostname. You can even specify multiple hostnames separated by a comma.

Working with inventory files

An inventory file is the source of truth for Ansible (there is also an advanced concept called **dynamic inventory**, which we will cover later). It follows the INI format and tells Ansible whether the remote host or hosts provided by the user are genuine or not.

Ansible can run its tasks against multiple hosts in parallel. To do this, you can directly pass the list of hosts to Ansible using an inventory file. For such parallel execution, Ansible allows you to group your hosts in the inventory file; the file passes the group name to Ansible. Ansible will search for that group in the inventory file and run its tasks against all the hosts listed in that group.

You can pass the inventory file to Ansible using the `-i` or `--inventory-file` option followed by the path to the file. If you do not explicitly specify any inventory file to Ansible, it will take the default path from the `host_file` parameter of `ansible.cfg`, which defaults to `/etc/ansible/hosts`.

The basic inventory file

Before diving into the concept, first let's look at a basic inventory file in the following screenshot:

```
$ cat hosts
example.com
web001
db001
50.16.4.125
```

Ansible can take either a hostname or an IP address within the inventory file. In the preceding example, we specified four servers; Ansible will take these servers and search for the hostname that you provided, to run its tasks. If you want to run your Ansible tasks against all of these hosts, then you can pass `all` to the hosts parameter while running the `ansible-playbook` or to the `ansible` command; this will make Ansible run its tasks against all the hosts listed in an inventory file.

The command that you would run is shown in the following screenshot:

```
$ ansible all -i hosts  -m ping -u ec2-user
example.com | success >> {
    "changed": false,
    "ping": "pong"
}

web001 | success >> {
    "changed": false,
    "ping": "pong"
}

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

db001 | success >> {
    "changed": false,
    "ping": "pong"
}
```

In the preceding screenshot, the Ansible command took all the hosts from an inventory file and ran the `ping` module against each of them. Similarly, you can use `all` with the `ansible-playbook` by passing `all` to the hosts. Let's see an example for an Ansible playbook in the following screenshot:

```
$ cat playbooks/setup_apache_all_hosts.yml
---
-   hosts: all
    remote_user: vagrant
    tasks:

      - name: Install httpd package
        yum: name=httpd state=latest
        sudo: yes

      - name: Starting httpd service
        service: name=httpd state=started
        sudo: yes

      - name: Show how debug works
        debug: msg={{ ansible_distribution }}
```

Now, when you run the preceding Ansible playbook, it will execute its tasks against all hosts listed in an inventory file.

This command will spawn off four parallel processes, one for each machine. The default number of parallel threads is five. For a larger number of hosts, you can increase the number of parallel processes with the `-f or --forks=< value >` option.

Coming back to the features of the file, one of the drawbacks with this type of simple inventory file is that you cannot run your Ansible tasks against a subset of the hosts, that is, if you want to run Ansible against two of the hosts, then you can't do that with this inventory file. To deal with such a situation, Ansible provides a way to group your hosts and run Ansible against that group.

Groups in an inventory file

In the following example, we grouped the inventory file into three groups, that is, `application`, `db`, and `jump`:

```
$ cat hosts
[application]
example.com
web001
[db]
db001
[jump]
192.168.2.1
```

Now, instead of running Ansible against all the hosts, you can run it against a set of hosts by passing the group name to the `ansible-playbook` command. When Ansible runs its tasks against a group, it will take all the hosts that fall under that group. To run Ansible against the `application` group, you need to run the command line shown in the following screenshot:

```
$ ansible application -i hosts  -m ping -u ec2-user
web001 | success >> {
    "changed": false,
    "ping": "pong"
}

example.com | success >> {
    "changed": false,
    "ping": "pong"
}
```

This time we directly passed the group name instead of running Ansible against all hosts; you can have multiple groups in the inventory file and you can even club similar groups together in one group (we will see how clubbing groups works in the next section). You can use groups using Ansible's `playbook` command as well by passing the group name to hosts.

```
$ cat playbooks/setup_apache_application_hosts.yml
---
-   hosts: application
    remote_user: vagrant
    tasks:

        - name: Install httpd package
          yum: name=httpd state=latest
          sudo: yes

        - name: Starting httpd service
          service: name=httpd state=started
          sudo: yes

        - name: Show how debug works
          debug: msg={{ ansible_distribution }}
```

In the preceding screenshot, Ansible will run its tasks against the hosts `example.com` and `web001`.

You can still run Ansible against a single host by directly passing the hostname or against all the hosts by passing `all` to them.

Groups of groups

Grouping is a good way to run Ansible on multiple hosts together. Ansible provides a way to further group multiple groups together. For example, let's say, you have multiple application and database servers running in the east coast and these are grouped as `application` and `db`. You can then create a master group called `eastcoast`. Using this command, you can run Ansible on your entire `eastcoast` data center instead of running it on all groups one by one.

Let's take a look at an example shown in the following screenshot:

```
$ cat hosts
[application]
example.com
web001
[db]
db001
[jump]
192.168.2.1
[eastcoast:children]
application
db
```

You can use a group of groups in the same way you use Ansible with groups in the preceding section. This is demonstrated in the following screenshot:

```
$ ansible eastcoast -i hosts -m ping -u ec2-user
example.com | success >> {
    "changed": false,
    "ping": "pong"
}

db001 | success >> {
    "changed": false,
    "ping": "pong"
}

web001 | success >> {
    "changed": false,
    "ping": "pong"
}
```

You can directly refer to an inventory group in the `ansible-playbook` as follows:

```
$ cat playbooks/setup_apache_eastcoast_hosts.yml
---
-   hosts: eastcoast
    remote_user: vagrant
    tasks:

      - name: Install httpd package
        yum: name=httpd state=latest
        sudo: yes

      - name: Starting httpd service
        service: name=httpd state=started
        sudo: yes

      - name: Show how debug works
        debug: msg={{ ansible_distribution }}
```

Regular expressions with an inventory file

An inventory file would be very helpful if you have many servers. Let's say you have a large number of web servers that follow the same naming convention, for example, `web001`, `web002`, …, `web00N`, and so on. Listing all these servers separately will result in a dirty inventory file, which would be difficult to manage with hundreds to thousands of lines. To deal with such situations, Ansible allows you to use `regex` inside its inventory file. The following screenshot shows an example of multiple servers:

```
$ cat hosts
[application]
web[001:200]
[db]
db[001:020]
[jump]
192.168.2.[1:3]
```

From the preceding screenshot, we can deduce the following:

- `web[001:200]` will match `web001`, `web002`, `web003`, `web004`, …, `web199`, `web200` for the `application` group
- `db[001:020]` will match `db001`, `db002`, `db003`, …, `db019`, `db020` for the `db` group
- `192.168.2.[1:3]` will match `192.168.2.1`, `192.168.2.2`, `192.168.2.3` for the `jump` group

External variables

Ansible allows you to define external variables in many ways, from an external variable file within a playbook, by passing it from the Ansible command using the `-e / --extra-vars` option, or by passing it to an inventory file. You can define external variables in an inventory file either on a per-host basis, for an entire group, or by creating a variable file in the directory where your inventory file exists.

Host variables

Using the following inventory file, you can access the variable `db_name` for the `db001` host, and `db_name` and `db_port` for `192.168.2.1`:
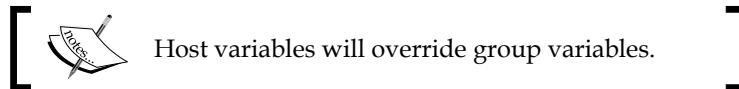
```
$ cat hosts
example.com
web001
db001 db_name=mysql
192.168.2.1 db_name=redis db_port=6380
```

Group variables

Let's move to variables that can be applied to a group. Consider the following example:

```
$ cat hosts
[application]
web[001:200]
[db]
db[001:020]
[jump]
192.168.2.[1:3]
[application:vars]
app_type=search
app_port=9898
```

The preceding inventory file will provide two variables and their respective values for the `application` group, `app_type=search` and `app_port=9898`.

> Host variables will override group variables.

Variable files

Apart from `host` and `group` variables, you can also have variable files. Variable files can be either for hosts or groups that reside in the folder of your inventory file. All of the host variable files will go under the `host_vars` directory, whereas all group variable files will go under the `group_vars` directory.

The following is an example of a host variable file (assuming your inventory file is under the `/etc/ansible` directory):
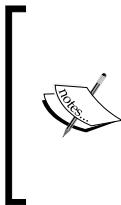
`cat /etc/ansible/host_vars/web001`

`app_type=search`

`app_port=9898`

As our inventory file resides under the `/etc/ansible` directory, we will create a `host_vars` directory inside `/etc/ansible` and place our variable file inside that. The name of the variable file should be the hostname, mentioned in your inventory file.

The following is an example of a group variable file:

`cat /etc/ansible/group_vars/db`

`db_name=redis`

`db_port=6380`

The variable file for groups is the same as the `host` file. The only difference here is that the variables will be accessible to all of the hosts of that group. The name of the variable file should be the group name, mentioned in your inventory file.

> Inventory variables follow a hierarchy; at the top of this is the common variable file (we discussed this in the previous section, *Working with inventory files*) that will override any of the host variables, group variables, and inventory variable files. After this, comes the host variables, which will override group variables; lastly, group variables will override inventory variable files.

Overriding configuration parameters with an inventory file

You can override some of Ansible's configuration parameters directly through the inventory file. These configuration parameters will override all the other parameters that are set either through `ansible.cfg`, environment variables, or passed to the `ansible-playbook`/`ansible` command. The following is the list of parameters you can override from an inventory file:

- `ansible_ssh_user`: This parameter is used to override the user that will be used for communicating with the remote host.

- `ansible_ssh_port`: This parameter will override the default SSH port with the user-specified port. It's a general, recommended `sysadmin` practice to not run SSH on the standard port 22.

- `ansible_ssh_host`: This parameter is used to override the host for an alias.

- `ansible_connection`: This specifies the connection type that should be used to connect to the remote host. The values are `SSH`, `paramiko`, or `local`.

- `ansible_ssh_private_key_file`: This parameter will override the private key used for SSH; this will be useful if you want to use some specific keys for a specific host. A common use case is if you have hosts spread across multiple data centers, multiple AWS regions, or different kinds of applications. Private keys can potentially be different in such scenarios.

- `ansible_shell_type`: By default, Ansible uses the `sh` shell; you can override this using the `ansible_shell_type` parameter. Changing this to `csh`, `ksh`, and so on will make Ansible use the commands of that shell.

- `ansible_python_interpreter`: Ansible, by default, tries to look for a Python interpreter within `/usr/bin/python`; you can override the default Python interpreter using this parameter.

Let's take a look at the following example:

```
example.com
web001 ansible_ssh_user=myuser ansible_ssh_private_key_file=myuser.rsa
db001
192.168.2.1
```

The preceding example will override the SSH user and the SSH private keys for the `web001` host. You can set similar variables for groups and variable files.

Working with modules

Now that we've seen how playbooks, variables, and inventories come together, let's look at modules in greater detail. Ansible provides more than 200 modules under top-level modules, such as `System`, `Network`, `Database`, and `Files`, that you can readily use and deal with in your infrastructure. The module index page has more details regarding the categories of the module. We'll explore modules that are commonly used and look at more advanced modules in later chapters.

Command modules

We start with four modules that are pretty similar to each other. They are used to execute tasks on remote machines. We need to take care of idempotency for most tasks that involve any of the above modules using conditional clauses that we will see in the coming chapters. Using parameters such as `creates` and `removes` can also introduce idempotency. Let's see when and where we can use each of these.

The command module

This takes the `command` name along with the arguments. However, shell variables or operations such as `<`, `>`, `|`, and `&` will not work as they will not be processed by the shell. This feature is similar to the `fork` function in C programming. Running the `command` module is secure and predictable. Also, the `command` module gives you the following parameters:

- `chdir`: This is used to change to a specific directory and execute commands
- `creates`: You can specify what file will be created with this option
- `removes`: This is used to remove a file

Let's write a task to reboot a machine, as follows:

```
- name: Reboot machine
  command: /sbin/shutdown -r now
  sudo: yes
```

On running the preceding command, we can see the following output:

```
TASK: [Reboot machine] **********************************************************
<web001> REMOTE_MODULE command /sbin/shutdown -r now
changed: [web001] => {"changed": true, "cmd": ["/sbin/shutdown", "-r", "now"], "delta": "0:00:00.
029714", "end": "2014-07-16 16:09:39.937290", "rc": 0, "start": "2014-07-16 16:09:39.907576", "st
derr": "", "stdout": ""}
```

As expected, without the conditional clause, this task will execute every single time as part of running the playbook.

The raw module

This module is used only when the other command modules do not fit the bill. This can be applied to a machine and will run a command in SSH. Use cases include running remote tasks on machines that don't have Python installed. Networking devices, such as routers and switches, are classic cases. Let's look at a quick example to install the vim package, as follows:

```
- name: Install vim
  raw: yum -y install vim-common
  sudo: yes
```

On running the preceding command, we see that the package is installed. Even after the package is installed, the task does not show that it is a changed task. It's best to not use the raw package.

```
TASK: [raw yum -y install vim-common] ******************************************
ok: [192.168.33.10]
```

The script module

This module is used to copy a script remotely to a machine and execute it. It supports the `creates` and `removes` parameters. Let's look at an example where we list down directories within a particular directory. Remember, you don't have to copy the script remotely in this case. The module does it for you as follows:

```
- name: List directories in /etc
  script: list_number_of_directories.sh /etc
  sudo: yes
```

On running the preceding command, we get the following output:

```
TASK: [List directories in /etc] **********************************************
changed: [web001] => {"changed": true, "rc": 0, "stderr": "", "stdout": "83\r\n"}
```

Here, `83` is the number of directories in the `/etc` directory, which can be verified by running the following command:

```
$ls -l /etc | egrep '^d' | wc -l
83
```

The shell module

Finally we come to the `shell` module. The major difference between the `command` and `shell` modules is that the `shell` module uses a shell (`/bin/sh`, by default) to run the commands. You can use shell environment variables and other shell features. Let's look at an example where we redirect the list of all files in `/tmp` to a directory and, in a subsequent task, concatenate (using `cat`) the file. The tasks are shown as follows:

```
- name: List files in /tmp and redirect to a file
  shell: /bin/ls -l /tmp > /tmp/list
  sudo: yes

- name: Cat /tmp/list
  shell: /bin/cat /tmp/list
```

The output is shown as follows:

```
TASK: [List files in /tmp and redirect to a file] *****************************
<web001> REMOTE_MODULE command /bin/ls -l /tmp > /tmp/list #USE_SHELL
changed: [web001] => {"changed": true, "cmd": "/bin/ls -l /tmp > /tmp/list", "delta": "0:00:00.
014685", "end": "2014-10-13 18:17:09.343955", "rc": 0, "start": "2014-10-13 18:17:09.329270", "
stderr": "", "stdout": ""}

TASK: [Cat /tmp/list] *****************************************************
<web001> REMOTE_MODULE command /bin/cat /tmp/list #USE_SHELL
changed: [web001] => {"changed": true, "cmd": "/bin/cat /tmp/list", "delta": "0:00:00.013115",
"end": "2014-10-13 18:17:16.634651", "rc": 0, "start": "2014-10-13 18:17:16.621536", "stderr":
"", "stdout": "total 0\n-rw-r--r-- 1 root root 0 Oct 13 18:17 list"}
```

> We've turned off color for screenshots that involve debugging just to make them more readable. The preceding output might not look that great but it can be formatted. Using `callbacks` and `register`, you can format an output in a better manner. We'll demonstrate that in later chapters.

File modules

We'll now switch to some very useful file modules, namely, `file`, `template`, and `copy`. There are others as well but we intend to cover the most important ones and those that are used often. Let's start with the `file` module.

The file module

The `file` module allows you to change the attributes of a file. It can touch a file, create or delete recursive directories, and create or delete symlinks.

The following example makes sure that `httpd.conf` has the right permissions and owner:

```
- name: Ensure httpd conf has right permissions and owner/group
  file: path=/etc/httpd/conf/httpd.conf owner=root group=root mode=0644
  sudo: yes
```

On running the preceding command, you should see the following output:

```
TASK: [Ensure httpd conf has right permissions and owner/group] ***************
ok: [web001]
```

If we check the output on the machine, we will see the following:

```
$ ls -l /etc/httpd/conf/httpd.conf
-rw-r--r-- 1 root root 34474 Sep 15 19:40 /etc/httpd/conf/httpd.conf
```

As shown in the preceding screenshot, there is no change as the file has the expected permissions and ownership. However, it's important to make sure important configuration files are under the control of Ansible (or any configuration management tool in general) so that, if there are changes, the next time playbook is run, those changes are reverted. This is one of the reasons for having your infrastructure as code, making sure you control all that matters from the code that is checked in. If there are genuine changes, then those have to be checked into the main Ansible repository that you maintain, and change has to flow from there.

The next example will create a symlink to the `httpd conf` file, as follows:

```
  - name: Create a symlink in /tmp for httpd.conf
    file: src=/etc/httpd/conf/httpd.conf dest=/tmp/httpd.conf owner=root
group=root state=link
    sudo: yes
```

The output of the preceding task is as follows:

```
TASK: [Create a symlink in /tmp for httpd.conf] ********************************
changed: [web001]
```

If we check on the machine, we will see the following output:

```
$ ls -l /tmp/httpd.conf
lrwxrwxrwx 1 root root 26 Oct 13 18:24 /tmp/httpd.conf -> /etc/httpd/conf/httpd.conf
```

The output is as expected. You might notice that we're running the `ls` command to verify the output. This is not always necessary, but it's highly recommended that you test everything that you automate right from the start. In the next chapter, we'll show you the possible methods in which you can automate these tests. For now, they are manual.

Debugging in Ansible

Now, let's create a hierarchy of directories with 777 permissions. For this particular example, we're going to use Ansible 1.5.5 for the purpose of showcasing how to debug with Ansible:

```
- name: Create recursive directories
  file: path=/tmp/dir1/dir2/dir3 owner=root group=root mode=0777
  sudo: yes
```

Do you see something not right with the preceding example? (In hindsight, if we've asked you the question, it means something is wrong!)

Let's run it and check. On running it, we see the following output:

```
TASK: [Create recursive directories] *****************************************
ok: [192.168.33.10]
```

We would expect this task's output to be `changed`. However, it shows `ok`. Let's verify it on the system.

```
$ ls -l /tmp/
total 48
lrwxrwxrwx  1 root     root        26 Jul 18 07:02 httpd.conf -> /etc/httpd/conf/httpd.conf
-rwx------. 1 root     root       244 Apr 27  2013 ks-script-FsgqMe
-rwxr-xr-x. 1 root     root       233 Apr 27  2013 ks-script-FsgqMe.log
-rw-r--r--  1 root     root       586 Jul 18 07:06 list
drwx------  2 vagrant vagrant   4096 Jul 18 07:01 ssh-JtWgIV5729
drwx------  2 vagrant vagrant   4096 Jul 18 08:22 ssh-WsNeVr7285
-rw-r--r--  1 root     root     23671 Apr 27  2013 stderr
-rw-------. 1 root     root         0 Apr 27  2013 yum.log
-rw-r--r--  1 root     root       536 Jul  4 17:41 yum.sh
```

There you go! The recursive directories are not present. Why did this happen without an error?

To find the answer, run the `-vv` option you learned about earlier. The following output is received:

```
TASK: [Create recursive directories] *****************************************
<192.168.33.10> REMOTE_MODULE file path=/tmp/dir1/dir2/dir3 owner=root group=root mode=0777
ok: [192.168.33.10] => {"changed": false, "path": "/tmp/dir1/dir2/dir3", "state": "absent"}
```

This was a bug in Version 1.5.5 but was fixed later in Version 1.6 and without specifying `state=directory`, it errors out. However, there is a possibility that you might find other such issues. Make sure you check the documentation; it might be a bug that you might want to raise or possibly fix. To fix the preceding bug in Version 1.5.5, we change the `state` value to `directory`, as shown in the following command lines.

```
- name: Create recursive directories
  file: path=/tmp/dir1/dir2/dir3 owner=root group=root mode=0777
state=directory
  sudo: yes
```

On running the preceding command line with the debug mode this time, we will see the following output:

```
TASK: [Create recursive directories] *****************************************
changed: [web001]
```

Looking at the tree output on the machine, we see that the directory has been created. This is depicted in the following screenshot:

```
$ tree /tmp/
/tmp/
|-- dir1
|   `-- dir2
|       `-- dir3
|-- httpd.conf -> /etc/httpd/conf/httpd.conf
|-- ks-script-FsgqMe
|-- ks-script-FsgqMe.log
|-- list
|-- ssh-JtWgIV5729
|   `-- agent.5729
|-- stderr
|-- yum.log
`-- yum.sh
```

The moral of the story is, Learn debugging techniques with a tool so that you can resolve issues at the earliest!

Let's move to another very useful module, `template`. This is as useful as the `template` resource in Chef/Puppet.

The template module

Ansible uses the Jinja2 templating language for creating templates, modeled after Django's templates (Django is a popular Python web framework). This is similar to Erubis, which Puppet and Chef use.

Templating is also a way to create a file on a machine. Let's now create a simple template using the following command lines:

```
$cat test
This is a test file on {{ ansible_hostname }}
```

The test file is in the same directory as `example1.yml`.

To reference the template, we'll add the following to the playbook:

```
  - name: Create a test template
    template: src=test dest=/tmp/testfile mode=644
```

On running the preceding playbook, we get the following output:

```
TASK: [Create a test template] *******************************************
****
changed: [192.168.33.10] => {"changed": true, "dest": "/tmp/testfile", "gid
": 501, "group": "vagrant", "md5sum": "d285bafc32d4bbba6f6beab9118d0105", "
mode": "0644", "owner": "vagrant", "size": 37, "src": "/home/vagrant/.ansib
le/tmp/ansible-tmp-1405675400.96-142249486362906/source", "state": "file",
"uid": 501}
```

As you can see in the following screenshot, Ansible created `testfile` inside `/tmp` and applied the template to the file.

```
$ ls /tmp/testfile
/tmp/testfile
$ cat /tmp/testfile
This is a test file on 192.168.33.10
$
```

The user running the playbook is `vagrant` in this case and the file created will also be owned by the same user. The `ansible_hostname` variable is populated during the `gather facts` phase. Let's take a minor detour and disable `gather facts` by adding the following to the playbook:

```
- hosts: host1
  gather_facts: False
```

Now, on running the playbook, the debug task fails as follows:

```
TASK: [Show how debug works] **********************************************
fatal: [192.168.33.10] => One or more undefined variables: 'ansible_distribution' is
 undefined

FATAL: all hosts have already failed -- aborting
```

On commenting out the task and running it again, we get an error with the template, as shown in the following screenshot:

```
TASK: [Create a test template] ********************************************
fatal: [192.168.33.10] => {'msg': "One or more undefined variables: 'ansible_hostname'
 is undefined", 'failed': True}
fatal: [192.168.33.10] => {'msg': "One or more undefined variables: 'ansible_hostname'
 is undefined", 'failed': True}

FATAL: all hosts have already failed -- aborting
```

Now, there are several cases where you might not want to gather facts. In such cases, to refer to the host, Ansible provides another useful variable `inventory_hostname`, which you can use. To modify the template, use the following command line:

```
cat playbooks/test
This is a test file on {{ inventory_hostname }}
```

On deleting the test file and rerunning the Ansible playbook, we find the same result as before:

```
TASK: [Create a test template] **********************************************
changed: [192.168.33.10] => {"changed": true, "dest": "/tmp/testfile", "gid": 501
, "group": "vagrant", "md5sum": "d285bafc32d4bbba6f6beab9118d0105", "mode": "0644
", "owner": "vagrant", "size": 37, "src": "/home/vagrant/.ansible/tmp/ansible-tmp
-1405676409.55-101916036497719/source", "state": "file", "uid": 501}
```

As expected, Ansible created `testfile` and did not fail because we used the `inventory_hostname` variable this time:

```
$ cat /tmp/testfile
This is a test file on 192.168.33.10
```

The Ansible template also has a `validate` parameter that allows you to run a command to validate the file before copying it. This is like a hook that Ansible provides to make sure files that might break the service are not written. A classic example is that of the `Apache httpd` configuration. You can verify that the `Apache httpd` configuration files have the right syntax using the `httpd` or `apachectl` command. Since the `validate` command takes an argument, we'll use the `httpd` option. It works as follows:

```
$httpd -t -f /etc/httpd/conf/httpd.conf

Syntax OK
```

If we introduce an error by uncommenting the `Virtual hosts` line, we get the following output when we rerun the `$httpd -t -f /etc/httpd/conf/httpd.conf` command:

```
httpd: Syntax error on line 1003 of /etc/httpd/conf/httpd.conf: /etc/
httpd/conf/httpd.conf:1003: <VirtualHost> was not closed.
```

We'll demonstrate the same technique for a new virtual host file in Apache. We'd like to verify that the new virtual host file that we're adding has the right syntax. Let's look at the virtual host file. There is an error, as shown in the following screenshot:

```
$ cat playbooks/test.conf
# Ensure that Apache listens on port 80
# This is defined in the vars folder. Everything else is hardcoded for simplicity
Listen {{ port }}
# Listen for virtual host requests on all IP addresses
# ERROR *** Instead of NameVirtualHost *:80 we have ***
VirtualHost *:80
<VirtualHost *:80>
DocumentRoot /www/example1
ServerName www.example.com
</VirtualHost>
```

The playbook will have the following template task. The `validate` option takes a `%s` parameter, which is the source file that we're planning to write to the machine:

```
  - name: Create a virtual host
    template: src=test.conf dest=/etc/httpd/conf.d/test.conf mode=644
validate='httpd -t -f %s'
    sudo: yes
```

Now, on running the preceding command line, we get the following output:

```
TASK: [Create a virtual host] ************************************************
failed: [192.168.33.10] => {"failed": true}
msg: failed to validate: rc:1 error:Syntax error on line 6 of /home/vagrant/.ansi
ble/tmp/ansible-tmp-1405676774.6-215874791961233/source:
Invalid command 'VirtualHost', perhaps misspelled or defined by a module not incl
uded in the server configuration
```

Ansible points out the error and the file is not written. This is a great feature that Ansible templates offer; if we have scripts for different packages/services that can verify the validity of files, then we'll have a lot fewer breakages due to incorrect configuration files. It is especially useful to have the `validate` feature when you write your own modules. We will cover how to write custom modules in a later chapter.

Let's move to the next module, `copy`.

The copy module

Using `copy`, you can copy a file from your local location to remote machines. This is another way to create a remote file with predetermined content (the template being the first). Let's look at an example as follows:

```
 - name: Copy file remotely
   copy: src=test2.conf dest=/etc/test2.conf owner=root group=root
mode=0644
```

On running the preceding command, we see the following output:
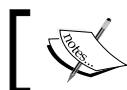
```
TASK: [Copy file remotely] ************************************************
failed: [192.168.33.10] => {"failed": true, "md5sum": "ffd0084c4de052b8bd423e2fc5
552837"}
msg: Destination /etc not writable

FATAL: all hosts have already failed -- aborting
```

Any idea why this failed? If you figured out that it's because `sudo: true` is not part of the module invocation, share a high five with the nearest person. Once we add it, the run goes through without errors, as shown in the following screenshot:

```
TASK: [Copy file remotely] ****************************************************
changed: [192.168.33.10]
```

The `copy` module also supports the `validate` parameter just like the `template` module.

> With simple modules, we've tried to highlight possible errors that you might come across during your experience with Ansible.

The source control module – git

We'll now look at a very important module, the source control module `git`. Ansible has modules to support `subversion`, `bzr`, and `hg`, apart from `github_hooks`. We'll look at possibly the most popular source control system today, `git`.

Let's check out a git repository from GitHub using an Ansible task. We'll first install `git` with a `yum` task, shown as follows:

```
- yum: name=git state=installed
  sudo: yes
```

Now, let's check out the popular `gitignore` repository. The Ansible task is shown as follows:

```
- name: Checkout gitignore repository
  git: repo=https://github.com/github/gitignore.git
       dest=/opt/gitignore
  sudo: yes
```

On running the playbook (using `-vv` here), we will see the following output:

```
TASK: [Checkout gitignore repository] ***************************************
<192.168.33.10> REMOTE_MODULE git repo=https://github.com/github/gitignore.git des
t=/opt/gitignore
changed: [192.168.33.10] => {"after": "21f1d7772141177dc76e1d1a5367ba18590b6169",
"before": null, "changed": true}
```

If we check the machine, we will see that the repository has been checked out in the expected directory.

```
$ ls -lh /opt/gitignore/ | wc -l
116
```

On rerunning the task, we see the following output:

```
TASK: [Checkout gitignore repository] ****************************************
<192.168.33.10> REMOTE_MODULE git repo=https://github.com/github/gitignore.git des
t=/opt/gitignore
ok: [192.168.33.10] => {"after": "21f1d7772141177dc76e1d1a5367ba18590b6169", "befo
re": "21f1d7772141177dc76e1d1a5367ba18590b6169", "changed": false}
```

This basically means that the task is idempotent as it checks the `before` and `after` **SHA** (**Secure Hash Algorithm**) values.

There are several other parameters with the git module, such as `depth` and `version` (which version to checkout). Quite often, we need to check out the git repository via the SSH protocol. The public keys have to be added to the repository and post that the checkout can happen. The git module has two other parameters, `accept_hostkey` and `key_file`, to help you in the git checkouts. The following is an example of a sample repository in one of our GitHub accounts, which we'll checkout on the remote machine. This example assumes that the private key pair is already present in `~/.ssh`.

Consider the following task for example:

```
- name: Checkout git repo via ssh
  git: repo=git@github.com:madhurranjan/node_nagios.git
       dest=/tmp/node_nagios
       accept_hostkey=yes
```

The output for the preceding command line is as follows:

```
TASK: [Checkout git repo via ssh] ****************************************
<192.168.33.10> REMOTE_MODULE git repo=git@github.com:madhurranjan/node_nagios.git
 dest=/tmp/node_nagios accept_hostkey=yes
changed: [192.168.33.10] => {"after": "306f00b58e4286e2a68a584e3052c31c4a67299f",
"before": null, "changed": true}
```

Summary

We end this chapter by summarizing what we've learned. We looked at an introduction to Ansible, wrote our very first playbook, learned ways to use the Ansible command line, learned how to debug playbooks, how Ansible does configuration management at a basic level, how to use variables, how inventory files work, and finally looked at modules. We hope you had a good time learning so far, which is why we recommend a coffee break before you start the next chapter! You can think of the following questions during your break to see whether you've understood the chapter:

- What does Ansible offer?
- How can variables be used in Ansible?
- What files in your environment can be templatized?
- How can you create inventory files for your current inventory?

We will look at more advanced modules and examples in later chapters. In the next chapter, we will look at how to develop Ansible playbooks and the best practices you should follow. We will focus on how to develop Ansible playbooks, test them, and the release cycle for the Ansible playbooks themselves. Make sure you do this in the right manner and incorporate best practices right from the beginning; this will save you a lot of time once your code matures.

2

Developing, Testing, and Releasing Playbooks

"Gone are the days when code to manage infrastructure was randomly scattered and deployed in Production."

- Anonymous

Quite often, you might have encountered situations wherein various teams introduce random tidbits of code in Production, in files that are stored in an obscure directory on the system. Quite often, you might also have teams suddenly chime in regarding certain scripts that were in the working state last week but is no longer the case this week. We've also seen situations where the operation team deals with developers in an unpleasant manner for introducing changes as part of releases that are not documented well enough. However, at the same time, they themselves release and use pieces of scripts that no one else knows about, akin to pulling a rabbit out of the hat.

Having seen these practices at close quarters, we firmly believe that these are more or less similar to voodoo practices that need to be done away with. Instead, we recommend that every operations/sysadmin team should follow a very similar flow when dealing with the infrastructure code in a way that development teams do (we're hoping the development teams actually follow the right procedure!) for all their products and services. In this chapter, we'll cover standard practices that most development teams follow and see how these can be applied to the infrastructure code keeping Ansible in mind.

In this chapter, we will cover the following topics:

- Managing source code
- Developing a playbook
- Testing a playbook
- The Serverspec tool
- Handling environments

Managing source code – Git

It is imperative that the right practices with respect to source code are followed right from day one. If you're a team of five people, for example, the best way to share code with each other is via a version control system. There are plenty of options, such as Git, SVN, Perforce, HG, and many others, but the most popular system and the one that we'll cover here is Git.

Before we look at how to use Git, we'd like to add a quick note on centralized versus distributed version control systems. Centralized version control systems work with a single central copy (which can also be a single point of failure) if not replicated. For larger repositories, every commit to the remote system is relatively slow. In the case of distributed version control systems, the entire repository, including the metadata, is present on the developer's machine. It supports the notion of local commits before you eventually push to the remote server. SVN and CVS fall in the centralized version control category, whereas Git and Mercurial fall in the distributed version control category. Now, let's jump into Git.

Git is a distributed version control system designed to handle everything, from small to very large projects, with speed and efficiency. Thanks to popular sites, such as GitHub and Bitbucket, the projects that use Git have grown in number rapidly. You can refer to the Git website (`http://git-scm.com/`) and learn more about the same. Assume that we have the `example1` playbook and its contents from the previous chapter. The tree structure of the `example1` playbook is as follows:

```
$ tree example1
.
├── hosts
└── playbooks
    └── example1.yml
```

One way to handle Git for Ansible is to use two branches: `dev` and `master`. By default, the branch that comes with Git is the `master` branch. We need to keep in mind the following rules:

- **Rule 1**: The `master` branch is always stable and should be treated as sacred
- **Rule 2**: All new commits have to be committed to the `dev` branch

Let's look at the basic command flow to illustrate the preceding tree flow of the `example1` playbook.

1. In the command window, run `git init .` after entering the `example1` directory, as follows:

   ```
   $ git init .
   Initialized empty Git repository in /Projects/ansible/example1/.git/
   ```

2. Let's add all the files that we have right now, (we've considered one of the examples from the first chapter) as the first commit, shown as follows:

   ```
   $ git add . ('.' represents the entire folder)
   ```

   To view the status of your directory at any stage, make sure you run `git status`. In other words, when in doubt, run `git status`, as follows:

   ```
   $ git status
   On branch master

   Initial commit

   Changes to be committed:
    (use "git rm --cached <file>..." to unstage)

           new file:    hosts
           new file:    playbooks/example1.yml
           new file:    playbooks/list_number_of_directories.sh
           new file:    playbooks/test
           new file:    playbooks/test.conf
           new file:    playbooks/test2.conf
   ```

3. We'll always commit first to the `dev` branch because, by default, it assumes you are in the `master` branch:

   ```
   $ git checkout -b dev
   Switched to a new branch 'dev'
   ```

4. Let's now commit what we have, as follows:

```
$ git commit -m "First commit"
[dev (root-commit) b3b9f27] First commit
6 files changed, 85 insertions(+)
create mode 100644 hosts
create mode 100644 playbooks/example1.yml
create mode 100644 playbooks/list_number_of_directories.sh
create mode 100644 playbooks/test
create mode 100644 playbooks/test.conf
create mode 100644 playbooks/test2.conf
```

5. Let's run a few basic Git commands and see what they do:

   $ git log ('git log' is another popular Git command to view the commit SHAs, the author of the commits, the date when it was committed, and the commit message shown as follows).

   commit b3b9f27c79600496458dcd5d7301c01dc3d29691

   Author: <author name> <email id>

   Date: Sun Jul 6 17:19:10 2014 +0530

   First commit

   $ git status (Now we see that there is nothing to commit).

   On branch dev

   nothing to commit, working directory clean

6. Now that we have committed our changes, we need a Git repository to push these changes to. Go ahead and create your repository as recommended in your organization, and add the Git link's remote URL, as follows:

   $ git remote add origin git@<remote link>:ansible_repository.git

7. Once the remote URL is added, we have to push the local changes to the remote branch, as follows:

   $ git push origin dev:dev

   This will create a remote branch named `dev`, corresponding to your local `dev` branch, and push your changes. We have now seen what we have to do to fulfill Rule 1, which is to make sure all the new commits go into the `dev` branch.

8. Now, assuming that the commits we have so far are stable (remember, we manually tested it after running Ansible playbooks in the first chapter), merge the `dev` branch to `master` and push it to a remote `master`. Now, run the following commands:

   ```
   $ git checkout -b master. (We have to add the '-b' here since
   there is no prior master branch.)


   $ git merge dev ( This merges all your dev branch into master).
   Without much ado, let's now push it to the remote master.


   $ git push origin master:master
   ```

   The preceding commands are carried out to fulfill Rule 2, which is to move all your changes from `dev` to `master` once they're stable.

If we enforce the preceding two rules amongst team members with discipline, we can make sure that the `master` branch is always stable. Discipline in this case, comes with a Continuous Integration job or pipeline that will run tests on each of the two branches. In a lot of cases, when a commit to `dev` branch results in a successful CI job or pipeline execution, it is automatically merged to `master`. In the later part of this chapter, we'll discuss how we can test our playbooks as part of CI or otherwise, and then deploy the preceding code to the command center or centers that we discussed in *Chapter 1*, *Getting Started with Ansible*. Till then, let's focus on other aspects such as playbook development.

> There are many other kinds of flows/branching strategies that people use for development, such as feature branches, early branching, and late branching, but that is out of the scope of this book. What we've covered here is a simple flow that you may follow but this is not necessarily binding on any team or individual.
>
> Also, from a Git perspective, there are other standard Git commands that are not covered here. The aim is to show a working model with basic commands. For more information regarding Git, please refer to any online Git tutorial or the Git link shared in the preceding section.

Developing a playbook

In Ansible, except for ad hoc tasks that are run using the `ansible` command, we need to make sure we have playbooks for every other repeatable task. In order to do that, it is important to have a local development environment, especially when a larger team is involved, where people can develop their playbooks and test them before checking them into Git.

A very popular tool that currently fits this bill is **Vagrant**. Vagrant's aim is to help users create and configure lightweight, reproducible, and portable development environments. By default, Vagrant works on VirtualBox, which can run on a local laptop or desktop. To elaborate further, it can be used for the following use cases:

- Vagrant can be used when creating development environments to constantly check new builds of software, especially when you have several other dependent components.

  For example, if I am developing service A and it depends on two other services, B and C, and also a database, then the fastest way to test the service locally is to make sure the dependent services and the database are set up (especially if you're testing multiple versions), and every time you compile the service locally, you deploy the module against these services and test them out.

- Testing teams can use Vagrant to deploy the versions of code they want to test and work with them, and each person in the testing team can have local environments on their laptop or desktop rather than common environments that are shared between teams.

- If your software is developed for cloud-based platforms and needs to be deployed on AWS and Rackspace (for example), apart from testing it locally on VMware Fusion or VirtualBox, Vagrant is perfect for this purpose. In Vagrant's terms, you can deploy your software on multiple providers with a configuration file that differs only for a provider.

  For example, the following screenshot shows the VirtualBox configuration for a simple machine:

```
Virtualbox:
Vagrant.configure("2") do |config|
  config.vm.box = "centos"
  config.vm.provider :virtualbox do |provider, override|
  end
  config.vm.define :test do |machine|
    machine.vm.hostname = "test"
    machine.vm.network "forwarded_port", guest: 22, host: 2220, id: "ssh", auto_correct: true
    machine.vm.network "private_network", ip: "192.168.33.11"
  end
end
```

The following is the AWS configuration for a simple machine:

```
AWS:
Vagrant.configure("2") do |config|
  config.vm.box = "aws"
  config.vm.provider :aws do |provider, override|
    provider.access_key_id = "<access key>"
    provider.secret_access_key = "<secret key>"
    provider.keypair_name = "<keypair name>"
    provider.instance_ready_timeout = 300  # seconds
    provider.instance_type = "< instance type>"
    provider.ami = "ami-94cd60fd"
    provider.region = "us-east-1"
    override.ssh.username = "ec2-user"
    override.ssh.private_key_path = "~/.ssh/DevopsAdminEast.pem"
  end
  config.vm.define :test do |machine|
    machine.vm.hostname = "test"
    machine.vm.network "forwarded_port", guest: 22, host: 2220, id: "ssh", auto_correct: true
  end
end
```

As you can see, the provider configuration changes but the rest of the configuration remains more or less the same. (Private IP is virtual-box-specific but it is ignored when run using the AWS provider.)

- Vagrant also provides **provisioners** (Vagrant provisioners will be explained shortly), which is what we're going to focus on. Vagrant provides users with multiple options to configure new machines as they come up using provisioners. They support shell scripts, tools such as Chef, Puppet, Salt, and Docker, as well as our focus in this book, Ansible.

By using Ansible with Vagrant, you can develop your Ansible scripts locally, deploy and redeploy them as many times as needed to get them right, and then check them in. The advantage, from an infrastructure point of view, is that the same code can also be used by other developers and testers when they spawn off their vagrant environments for testing (The software would be configured to work in the expected manner by Ansible playbooks.). The checked-in Ansible code will then flow like the rest of your software, through testing and stage environments before it is finally deployed into Production. So, let's look at how we can start using Ansible to develop the `example1.yml` file that we used in our previous chapter.

Installing VirtualBox and Vagrant

Install VirtualBox from the VirtualBox downloads page at `https://www.virtualbox.org/wiki/Downloads`. Verify that VirtualBox has been installed from the command line by running `vboxmanage --version`.

To install Vagrant, refer to the download link `http://www.vagrantup.com/downloads.html`. Download it for the distribution of your choice. On installing Vagrant, run `vagrant --version` to verify that the installation went through correctly.

Downloading the Vagrant box

The next step is to download a base box from which you can spawn machines. There are plenty of base boxes that can be used. For example, if you're running your Ansible scripts on Mac and would like to test them on Ubuntu 14.04 and CentOS 6.5, you need to download two base boxes corresponding to these versions and set them up. A good starting point to search for these images is `http://www.vagrantbox.es/`. You can also generate your own Vagrant box according to the instructions provided on the Vagrant website, `http://www.vagrantup.com/`.

Once you download the Vagrant box, you need to add it to the local Vagrant repository using `vagrant box add`:

```
$ vagrant box add centos65 boxes/centos65base.box

$ vagrant box list

centos65 (virtualbox)
```

Developing a machine

The following command creates a `Vagrantfile`, which is required to run Vagrant commands:

```
$ vagrant init
```

Change the box in the `Vagrantfile` from `config.vm.box = "base"` to `config.vm.box = "centos65"` (replace it with the box you have):

```
$ vagrant up
```

By default, Vagrant uses the VirtualBox provider and calls the VirtualBox API to start up the machine. Verify that the machine comes up by running `vagrant status`. The status should show `running` for this machine. Let's now log in to the machine:

$ vagrant ssh

All Vagrant boxes have the `vagrant` user set up as a `sudo` user. So you will be able to run any command either by using `sudo` or changing to the root user. Congratulations on spawning up your Vagrant instance!

Provisioning in Vagrant using an Ansible provisioner

Vagrant defines a separate term called **provisioning**, which runs the configuration management blocks or blocks that are plugged into the `Vagrantfile`, be it Ansible, Chef, Puppet, or shell scripts. The provision step is very useful to test your scripts without waiting to relaunch the machine. However, keep in mind that `vagrant up`, by default, runs provisioning on the machine.

Let's now see how you can use the Ansible provisioner with Vagrant. The following screenshot shows the `Vagrantfile` that we have:

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

# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
 config.vm.box = "centos"
 config.vm.network :private_network, ip: "192.168.33.10"
 config.vm.provision "ansible" do |ansible|
   ansible.inventory_path = "hosts"
   ansible.playbook = "playbooks/example1.yml"
   ansible.verbose = "-vv"
   ansible.sudo = true
   ansible.limit = 'all'
 end
end
```

Let's understand the `Vagrantfile` in more detail. The following lines are part of the file:

- `config.vm.box = "centos"`: This indicates that the Vagrant box used is CentOS

- `config.vm.network :private_network, ip: "192.168.33.10"`: This option specifies that a network interface will be created (using NAT) and the IP, `192.168.33.10`, is assigned to it

- `config.vm.provision "ansible" do |ansible|`: This option introduces the Ansible provisioner block. This is a ruby block. The options are explained as follows:

  ```
  ansible.inventory_path = "hosts" -  Inventory file that needs
  to be considered
  ansible.playbook = "playbooks/example1.yml" - what playbook to
  run
  ansible.sudo = true – enables sudo access during the playbook
  run
  ansible.limit = 'all' - means that the 'vagrant provision'
  command will affect all the machines that are under consideration.
  'ansible.limit'  by default is set to 'default' , which means
  that it will affect only the machine that this ansible block is
  part of. Since we have playbooks that we eventually will apply to
  a larger set of machines, setting 'ansible.limit' to all makes
  sense.
  ```

Since we said that we'll start looking at how to develop the playbook we eventually ended up with, in *Chapter 1, Getting Started with Ansible*, we start with a single task in `example1.yml`, as follows:

```
$ cat playbooks/example1.yml
---
- hosts: host1
  gather_facts: False
  remote_user: vagrant
  vars:
    port: 8080
  tasks:
    - name: Install httpd package
      yum: name=httpd state=latest
      sudo: yes
```

Now, let's run `vagrant up`. This command will bring up the machine and also run Ansible on it. This is shown in the following screenshot:

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'devbase'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2222 (adapter 1)
[default] -- 80 => 8080 (adapter 1)
[default] Booting VM...
[default] Waiting for machine to boot. This may take a few minutes...
[default] Machine booted and ready!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
[default] Running provisioner: ansible...

PLAY [host1] ************************************************************

TASK: [Install httpd package] ******************************************

changed: [192.168.33.10]

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

If you only want to bring up a machine and not run `vagrant provision`, you can run `vagrant up –no-provision`. Let's look at the output of `vagrant provision` as follows:

```
$ vagrant provision
[default] Running provisioner: ansible...

PLAY [host1] ************************************************************

TASK: [Install httpd package] ******************************************
ok: [192.168.33.10]

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

Since we're running `vagrant provision` for the second time (yes, first time it ran with `vagrant up`), the console shows the output as `ok=1 changed=0`. The actual provisioning was performed the first time, and the second provisioning run didn't bring about any change (as expected). This is similar to what you learned in *Chapter 1, Getting Started with Ansible*, about idempotency.

> Other helpful options of the Ansible provisioner are specified at
> `https://docs.vagrantup.com/v2/provisioning/ansible.html`.

Once we run `vagrant provision` for the first task (the `Install httpd package` task), we have a working flow. We then proceed with the following algorithm:

1. Add a new task.
2. Run `vagrant provision`.
3. If it fails, go back and debug.
4. If it succeeds, proceed to the next task.
5. Once all the tasks are complete, run `vagrant destroy` to destroy the machine.
6. Run `vagrant up` to make sure that the machine comes up from scratch and gets configured successfully.

It is recommended that you check in the `Vagrantfile` along with each logically separate playbook so that any custom settings that are set for a playbook are stored in the source control.

Testing a playbook

Like software code, testing infrastructure code is an all-important task. There should ideally be no code floating around in Production that has not been tested, especially when you have strict customer SLAs to meet, and this is true even for the infrastructure. In this section, we'll look at syntactic checks, testing without applying the code on the machines (the no-op mode), and functional testing for playbooks, which are at the core of Ansible that triggers the various tasks you want to perform on the remote hosts. It is recommended that you integrate some of these into your **Continuous Integration** (**CI**) system that you have for Ansible to test your playbooks better. We'll be looking at the following points:

1. Syntax checking
2. Checking the mode with and without `diff`

3. Functional testing, which can be done in the following ways:

- ° Assertions on the end state of the system
- ° Testing with tags
- ° Serverspec (a different tool, but can work wonderfully with Ansible)

Using the --syntax-check option

Whenever you run a playbook, Ansible first checks the syntax of the playbook file. If an error is encountered, Ansible will error out saying there was a syntax error and will not proceed unless you fix that error. This syntax checking is performed only when you run the `ansible-playbook` command. When writing a big playbook or if you have included task files, it might be difficult to fix all of the errors; this might end up wasting more time. In order to deal with such situations, Ansible provides a way to check your YAML syntax as you keep progressing with your playbook. Let's see how this works in the following example screenshot:

```
$ ansible-playbook -i hosts playbooks/setup_apache.yml --syntax-check

playbook: playbooks/setup_apache.yml
```

In the preceding screenshot, the `ansible-playbook` command checked the YAML syntax of the `setup_apache.yml` playbook and showed that the syntax of the playbook was correct. Let's look at the resulting errors from the invalid syntax in the playbook, in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/setup_apache.yml --syntax-check
ERROR: Syntax Error while loading YAML script, playbooks/setup_apache.yml
Note: The error may actually appear before this position: line 11, column 8

    - name: Starting httpd service
      service: name=httpd state=started
      ^
```

The errors displayed in the preceding screenshot show that there is an indentation error at `action: shell /bin/ls"` task. Ansible also gives you the line number, column number, and the file name where this error exists. This should definitely be one of the basic tests that you should run as part of your CI for Ansible.

The check mode

The check mode (also known as the dry run or no-op mode) will run your playbook in a no-operation mode, that is, it will not apply any changes to the remote host; instead, it will just show the changes that will be introduced when a task is run. Whether the check mode is actually enabled or not depends on each playbook. The last time we ran the check, we found 75 out of 253 modules that support check_mode. You can get to know which modules support check_mode using the following command:

```
$ cd ~/src/ansible/library
$ ag -l supports_check_mode=True | wc -l
75
$ find -type f | wc -l
253
```

This helps you test how your playbook will behave and check if there will be any failures before running it on your production server. You run a playbook in the check mode by simply passing the --check option to your ansible-playbook command.

Let's see how the check mode works with the setup_apache.yml playbook, as follows:

```
$ ansible-playbook -i hosts playbooks/example1.yml --check

PLAY [web001] ***********************************************************

GATHERING FACTS *********************************************************
ok: [web001]

TASK: [Install httpd package] ******************************************
ok: [web001]

TASK: [Check httpd service] ********************************************
changed: [web001]

TASK: [Show how debug works] *******************************************
ok: [web001] => {
    "msg": "Amazon"
}

PLAY RECAP ************************************************************
web001                     : ok=4    changed=1    unreachable=0    failed=0
```

In the preceding run, instead of making the changes on the remote host, Ansible highlighted all the changes that would have occurred during the actual run. From the preceding run, you can find that `httpd` was already installed on the remote host, because of which Ansible's exit message for that task was `ok`.

```
TASK: [Install httpd package] *****************************************
******

ok: [host1]
```

Whereas, with the second task, it found that httpd service was not running on the remote host.

```
TASK: [Check httpd service] *******************************************
******

changed: [host1]
```

When you run the preceding playbook again without the `check` mode enabled, Ansible will make sure that the service state is `running`.

Indicating differences between files using --diff

In the check mode, you can use the `--diff` option to show the changes that would be applied to a file. Let's see an example of how `--diff` works, in the following screenshot:

```
$ cat playbooks/diff_example.yml
---
- hosts: host1
  remote_user: ec2-user
  tasks:

    - name: Install httpd package
      yum: name=httpd state=latest
      sudo: yes

    - name: Check httpd service
      service: name=httpd state=started
      sudo: yes

    - name: Update httpd.conf file
      template: src=/Users/ramesh/httpd.conf_web dest=/etc/httpd/conf/httpd.conf
      sudo: yes
```

In the preceding playbook, we added a task, which will copy the http.conf file to the remote host at /etc/httpd/conf/httpd.conf.

> The --diff option doesn't work with the file module; you will have to use the template module only.

In the following example, Ansible compares the current file of the remote host with the source file; a line starting with + indicates that a line was added to the file, whereas - indicates that a line was removed.

```
$ ansible-playbook -i hosts playbooks/diff_example.yml --check --diff

PLAY [host1] ******************************************************************

GATHERING FACTS ***************************************************************

ok: [host1]

TASK: [Install httpd package] *************************************************

ok: [host1]

TASK: [Check httpd service] **************************************************

ok: [host1]

TASK: [Update httpd.conf file] ***********************************************
--- before: /etc/httpd/conf/httpd.conf
+++ after: /Users/ramesh/httpd.conf_web
@@ -1001,10 +1001,18 @@
 # The first VirtualHost section is used for requests without a known
 # server name.
 #
 #<VirtualHost *:80>
 #     ServerAdmin webmaster@dummy-host.example.com
 #     DocumentRoot /www/docs/dummy-host.example.com
 #     ServerName dummy-host.example.com
 #     ErrorLog logs/dummy-host.example.com-error_log
 #     CustomLog logs/dummy-host.example.com-access_log common
 #</VirtualHost>
+
+<VirtualHost *:80>
+     ServerAdmin webmaster@host1.com
+     DocumentRoot /www/docs/host1.com
+     ServerName host1.com
+     ErrorLog logs/host1.com-error_log
+     CustomLog logs/host1.com-access_log common
+</VirtualHost>

changed: [host1]

PLAY RECAP *******************************************************************
host1                      : ok=4    changed=1    unreachable=0    failed=0
```

> You can also use `--diff` without the `--check` option, which will allow Ansible to make the specified changes and show the difference between two files.

The `check` mode is a test step that can potentially be used as part of your CI tests to assert how many steps have changed as part of the run. The other case where you can use this feature, is the part of the deployment process to check what exactly will change when you run Ansible on that machine.

Functional testing in Ansible

Wikipedia says Functional Testing is a **quality assurance** (**QA**) process and a type of black-box testing that bases its test cases on the specifications of the software component under the test. Functions are tested by feeding them input and examining the output; the internal program structure is rarely considered. Functional testing is as important as code when it comes to infrastructure.

From an infrastructure perspective, with respect to functional testing, we test outputs of our Ansible runs on the actual machines. Ansible provides multiple ways to perform the functional testing of your playbook; let's look at some of the most commonly used methods. Please note that the rest of the chapter will consider basic examples to show you how you can test these playbooks. This is not an indication of how best we should write playbooks. These topics will be covered in future chapters.

Functional testing using Assert

The check mode will only work when you want to check whether a task will change anything on the host or not. This will not help when you want to check whether the output of your module is what you expected. For example, let's say you wrote a module that will check if a port is up or not. In order to test this, you might need to check the output of your module and see whether it matches the desired output or not. To perform such tests, Ansible provides a way where you can directly compare the output of a module with the desired output.

Let's see how this works. Consider the following screenshot where the `assert_example.yml` file is executed:

```
$ cat playbooks/assert_example.yml
---
- hosts: web001
  remote_user: ec2-user
  tasks:
    - action: shell /bin/ls
      register: list_files

    - assert:
        that:
          - "'testfile.txt' in list_files.stdout_lines"
```

In the preceding playbook, we're running the `ls` command on the remote host and registering the output of that command in the `list_files` variable. (We'll cover more of the register functionality in the next chapter.)

Further, we ask Ansible to check whether the output of the `ls` command has the expected result. We do this using the `assert` module, which uses some conditional checks to verify if the `stdout` value of a task meets the expected output of the user. Let's run the preceding playbook and see what output Ansible returns, as shown in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/assert_example.yml -vv

PLAY [web001] ********************************************************

GATHERING FACTS *****************************************************
<web001> REMOTE_MODULE setup
ok: [web001]

TASK: [shell /bin/ls] ***********************************************
<web001> REMOTE_MODULE command /bin/ls #USE_SHELL
changed: [web001] => {"changed": true, "cmd": "/bin/ls", "delta": "0:00:00.017252", "end": "2014-10-13 17:44:28.790193"
, "rc": 0, "start": "2014-10-13 17:44:28.772941", "stderr": "", "stdout": "bin\ntestfile.txt"}

TASK: [assert ] *****************************************************
ok: [web001] => {"msg": "all assertions passed"}

PLAY RECAP **********************************************************
web001                     : ok=3    changed=1    unreachable=0    failed=0
```

We ran the preceding playbook in the verbose mode using the `-vv` option. In the first task, Ansible gave a list of all the files in the current directory, as shown in the following screenshot, and we stored that output in the `list_files` variable:

```
TASK: [shell /bin/ls] ***********************************************************
<web001> REMOTE_MODULE command /bin/ls #USE_SHELL
changed: [web001] => {"changed": true, "cmd": "/bin/ls", "delta": "0:00:00.017252", "end": "2014-10-13 17:44:28.790193"
, "rc": 0, "start": "2014-10-13 17:44:28.772941", "stderr": "", "stdout": "bin\ntestfile.txt"}
```

In the second task, Ansible checked whether the output stored in the `list_files` variable has a string `testfile.txt` (that is, we simply checked whether `testfile.txt` exists). Well, the output is a positive one, which is shown in the following screenshot:

```
TASK: [assert ] *****************************************************************
ok: [web001] => {"msg": "all assertions passed"}
```

The task passed with an `ok` message as `testfile.txt` was present in the `list_files` variable. Likewise, you can match multiple strings in a variable or multiple variables using the `and` and `or` operators (we will discuss these operators in the next chapter).

The `assertion` feature is quite powerful, and users who have written either unit or integration tests in their projects will be quite happy to see this feature!

Testing with tags

Tags are a great way to test a bunch of tasks without running an entire playbook. We can use tags to run actual tests on the nodes to verify the state that the user intended to be in, in the playbook. We can treat this as another way to run integration tests for Ansible on the actual box. The tag method to test can be run on the actual machines where you run Ansible, and also, it can be used primarily during deployments to test the state of your end systems.

In this section, we'll first look at how to use tags in general, their features that can possibly help us, not just with testing but even otherwise, and finally for testing purposes.

To add tags in your playbook, use the `tags` parameter followed by one or more tag names separated by commas. Let's see an example playbook with tagging, as shown in the following screenshot:

```
$ cat playbooks/tags_example.yml
---
- hosts: web001
  remote_user: ec2-user
  tasks:
    - name: Stop httpd service
      service: name=httpd state=stopped
      tags:
        - stop
      sudo: yes

    - name: Start httpd service
      service: name=httpd state=started
      tags:
        - start
      sudo: yes
```

In the preceding playbook, we added two tags: one to stop a service and one to start a service. You can now simply pass the `stop` tag to the `ansible-playbook` command and Ansible will run all tasks that are tagged as `stop`.

> You can still run an entire playbook by not passing any tags to the `ansible-playbook` command.

Now, in the following screenshot, you can see the output of a task that is tagged as `stop`:

```
$ ansible-playbook -i hosts playbooks/tags_example.yml --tags stop

PLAY [web001] ***********************************************************

GATHERING FACTS *********************************************************
ok: [web001]

TASK: [Stop httpd service] **********************************************
changed: [web001]

PLAY RECAP **************************************************************
web001                     : ok=2    changed=1    unreachable=0    failed=0
```

As expected, only the task tagged as `stop` ran. Let's see how multiple tags work in the following screenshot:

```
$ cat playbooks/multiple_tags_example.yml
---
- hosts: web001
  remote_user: ec2-user
  tasks:
    - name: Testing user sudo privilege
      command: /usr/bin/sudo -v
      register: sudo_response
      ignore_errors: yes
      tags:
        - pre_check

    - name: Stop further if User doesn't have sudo privilege
      fail: msg="User doesn't have sudo privilege."
      when: sudo_response.rc != 0
      tags:
        - pre_check

    - name: Stop httpd service
      service: name=httpd state=stopped
      tags:
        - stop
      sudo: yes

    - name: Start httpd service
      service: name=httpd state=started
      tags:
        - start
      sudo: yes
```

In the preceding playbook, we checked whether the user had `sudo` privileges or not. If not, the Ansible run will fail saying `User doesn't have sudo privilege`. We tagged this task as `pre_check` because this is something you might want to check before actually trying to start/stop a service.

The `Testing user sudo privilege` task in this example will run a `/usr/bin/sudo -v` command on a remote host and store its output in the `sudo_response` variable. The next task, that is, the `Stop if User doesn't have sudo privilege` task, will actually check if a user has `sudo` privileges or not. To do so, we will use the `when` condition, which checks the return code of the previous task. Now, let's run the preceding playbook with two tags: `pre_check` and `start`.

```
$ ansible-playbook -i hosts playbooks/multiple_tags_example.yml --tags pre_check,start

PLAY [web001] **********************************************************

GATHERING FACTS *******************************************************
ok: [web001]

TASK: [Testing user sudo privilege] **********************************
changed: [web001]

TASK: [Stop further if User doesn't have sudo privilege] *************
skipping: [web001]

TASK: [Start httpd service] ******************************************
changed: [web001]

PLAY RECAP ***********************************************************
web001                     : ok=3    changed=2    unreachable=0    failed=0
```
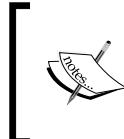
You can see in the preceding Ansible run that the `Stop if User doesn't have sudo privilege` task was skipped. This is because the user, `ec2-user`, on the host, `web001`, has `sudo` access. By passing tags, Ansible ran all the tasks that were tagged as `pre_check` and `start`. Like this, you can have multiple tags assigned to a task and call multiple tags from your `ansible-playbook` command.

You can also use tags to perform a set of tasks on the remote host just like taking a server out of a load balancer and adding it back to the load balancer.

> You can also use the `--check` option with tags. By doing this, you can test your tasks without actually running them on your hosts. This allows you to test a bunch of individual tasks directly, instead of copying your tasks to a temporary playbook and running it from there.

The --skip-tags

Ansible also provides a way to skip a couple of tags in a playbook. If you have a long playbook with multiple tags, say more than 10, then it would not be a good idea to pass 10 tags to Ansible. The situation would be more difficult if you miss to pass a tag and the `ansible-run` command fails. To overcome such situations, Ansible provides a way to skip a couple of tags, instead of passing multiple tags, which should run. Let's see how this works in the `multiple_tags_example.yml` playbook. The output is shown in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/multiple_tags_example.yml --skip-tags stop

PLAY [web001] *********************************************************************

GATHERING FACTS ******************************************************************
ok: [web001]

TASK: [Testing user sudo privilege] *********************************************
changed: [web001]

TASK: [Stop further if User doesn't have sudo privilege] ************************
skipping: [web001]

TASK: [Start httpd service] *****************************************************
ok: [web001]

PLAY RECAP *********************************************************************
web001                     : ok=3    changed=1    unreachable=0    failed=0
```

The preceding playbook will skip tasks that are tagged as `stop` and run the rest of the tasks.

Now that we've seen how a single or multiple tags work, we can use tags intelligently in our code to test all those tasks that we consider important. At the end of the run, we can run the same playbook with the `--tags` test, to check if the expected results were met, or you can write one test playbook for each playbook. For example, if your playbook is called `installation.yml`, your test playbook will be called `installation_test.yml`.

You might ask why we would need a task like this. A classic example is that of a `service` task, where you expect a service to start. Now, during the Ansible run, the playbook would have run `service <service name> start` and also shown that it started without any issues. However, if the `start` script itself calls another script, which has dependencies on another service/database, there is a good chance that the service start would fail. Running `service < service name> status` shortly afterwards would show that the service hasn't started.

Let's create two playbooks: `installation.yml` and `installation_test.yml`. The first playbook, `installation.yml`, will install `httpd service`, whereas `installation_test.yml` will check if `httpd` was installed on the remote host. The following screenshot demonstrates this task:

```
$ cat playbooks/installation.yml
---
- hosts: web001
  remote_user: ec2-user
  gather_facts: no

  tasks:
    - name: install httpd service
      yum: name=httpd state=latest
      tags:
        - install
      sudo: yes

    - name: Stop httpd service
      service: name=httpd state=stopped
      tags:
        - stop
      sudo: yes

    - name: Start httpd service
      service: name=httpd state=started
      tags:
        - start
      sudo: yes
```

The preceding playbook has three tags; they are as follows:

- `install`: This tag will install the `httpd` service
- `start`: This tag will start the `httpd` service
- `stop`: This tag will stop the `httpd` service

Using the install, start, and stop tags

You can install `httpd` by passing the `install` tag to the `ansible-playbook` command; similarly, you can start and stop the `httpd` service by passing the `start` and `stop` tags, respectively. Let's see how the `install` tag works, as shown in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/installation.yml --tags install

PLAY [web001] ********************************************************************

TASK: [install httpd service] **************************************************
changed: [web001]

PLAY RECAP *********************************************************************
web001                     : ok=1    changed=1    unreachable=0    failed=0
```

You can see that in the preceding `ansible-playbook` run, Ansible installs `httpd` on `web001`. Now, let's test if it was actually installed by Ansible. To test this, we first write our `installation_test.yml` playbook, as follows:

```
$ cat playbooks/installation_test.yml
---
- hosts: web001
  remote_user: ec2-user

  tasks:
    - name: Check if httpd is installed
      shell: rpm -qa | grep httpd
      tags:
        - install
      register: check_httpd
      sudo: yes

    - name: Notify if httpd is installed
      debug: msg="httpd is installed on {{ inventory_hostname }}"
      when: check_httpd.rc == 0
      tags:
        - install

    - name: Get status of httpd service
      command: /etc/init.d/httpd status
      register: httpd_status
      sudo: yes
      ignore_errors: true
      tags:
        - start
        - stop

    - name: Check if httpd service is stopped
      debug: msg="{{ httpd_status.stdout }}"
      tags:
        - stop
      sudo: yes

    - name: Check if httpd service is running
      debug: msg="{{ httpd_status.stdout }}"
      tags:
        - start
      sudo: yes
```

The preceding Ansible playbook has three tags: `install`, `start`, and `stop`. The tasks that are tagged as `install` will check to see whether the `httpd` service is installed. We do this by running the `rpm -qa | grep httpd` command on the remote host. Tasks tagged as `start` will check if the `httpd` service is running or not, and the `stop` tag will check if the `httpd` service is stopped. Let's run this playbook with an `install` tag and see if `httpd` was installed by our previous Ansible run. This is demonstrated in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/installation_test.yml --tags install

PLAY [web001] ***********************************************************

GATHERING FACTS *********************************************************
ok: [web001]

TASK: [Check if httpd is installed] ************************************
changed: [web001]

TASK: [Notify if httpd is installed] **********************************
ok: [web001] => {
    "msg": "httpd is installed on web001"
}

PLAY RECAP ************************************************************
web001                     : ok=3    changed=1    unreachable=0    failed=0
```

Similarly, you can test other steps as part of your playbook using tags.

The Serverspec tool

Serverspec (Serverspec Version 2, which was released recently, is called **Specinfra 2**) is another standalone tool, written in Ruby, which tests the intended state of machines by SSH'ing to the machines and running commands. Specinfra 2 supports only **rspec 3**, which is a popular Ruby test framework, as against rspec 2, which Serverspec v1 supported. Serverspec has gained tremendous popularity over the last several months and its USPs are that it can run with or without any of the configuration management tools, and it is agentless and declarative in nature. A typical workflow though is to run configuration management scripts followed by Serverspec tests.

Let's look at how to install Serverspec and how to start using it.

Installing Serverspec

If you're using the popular bundler gem in Ruby, add the `serverspec` gem to your `Gemfile` and run `bundle`; else, install it by just running `gem install serverspec`. The tests should be placed under a folder that matches the target hostname, role, or playbook. These can be tweaked with a `Rakefile` that is used for the tests. The `Rakefile` should be right at the root of the directory from where you intend to run the tests.

Let's see how we can write a simple test file to test check if `Apache httpd` has been installed, the service is up and enabled, and the port is running. The assumption is that `Apache httpd` has already been installed and is up and running before we run the following commands:

```
$ cat spec/example1/httpd_spec.rb

require 'spec_helper'
# This checks if package httpd is installed
describe package('httpd') do
    it { should be_installed }
end

# This checks if service is enabled and running
describe service('httpd') do
    it { should be_enabled  }
    it { should be_running  }
end

# This checks the port httpd is running on
describe port(80) do
    it { should be_listening }
end

# This checks for existence of httpd.conf
describe file('/etc/httpd/conf/httpd.conf') do
   it { should be_file }
end
```

Serverspec (both versions) requires a helper file named `spec_helper`, which provides information regarding what machines are present and how you can log in to those machines, including the SSH keys that are required for the same. A sample `spec_helper` that we've used for the tests is shown as follows:

```
$ cat spec/spec_helper.rb
#!/usr/bin/env ruby
require 'rubygems'
require 'serverspec'
require 'pathname'
require 'net/ssh'
#require 'json'

include Serverspec::Helper::Ssh
include Serverspec::Helper::DetectOS

RSpec.configure do |c|
  c.host  = ENV['TARGET_HOST']
  puts "C HOST --- #{c.host}"
  options = Net::SSH::Config.for(c.host)
  #can have all possible keys if you're using keys
  options[:keys] = ["~/.ssh/ansible_key"]
  options[:paranoid] = false
  user    = 'vagrant'
  c.ssh   = Net::SSH.start(c.host, user, options)
  c.os    = backend.check_os
end
```

Finally, we make sure that we provide the right inputs in terms of what machines to run the tests against by changing the values in the `Rakefile`. The following screenshot shows the `Rakefile`:

```
require 'rake'
require 'rspec/core/rake_task'
#require 'json'
hosts = [
  {
    :name  => 'host1',
    :ip    => '192.168.33.10',
    :playbook => %w( example1 ),
  },
]

desc "Run serverspec on all hosts"
task :serverspec => 'serverspec:all'

namespace :serverspec do
  #If you have a lot of hosts and want to run it in parallel
  #multitask :all => hosts.map {|h| 'serverspec:' + h[:name] }
  task :all => hosts.map {|h| 'serverspec:' + h[:name] }
  hosts.each do |host|
    desc "Run serverspec to #{host[:name]}"
    RSpec::Core::RakeTask.new(host[:name].to_sym) do |t|
      ENV['TARGET_HOST'] = host[:ip]
      puts "---------------------------------"
      puts "Running #{host[:playbook]} serverspec tests on - #{host[:name]} "
      t.pattern = "spec/#{host[:playbook]}/*_spec.rb"
      puts "hostname/ip - #{host[:name]}"
    end
  end
end
```

Hosts can have multiple nodes here. If the Ansible playbook has multiple nodes, then you'll have to test multiple nodes.

Let's now look at the structure of the `example1` playbook along with Serverspec tests, as follows:

```
$ tree
├── playbooks
│   ├── example1.yml
│   ├── list_number_of_directories.sh
│   ├── test
│   ├── test2.conf
│   └── test.conf
```

```
├── Rakefile
├── spec
│   ├── example1
│   │   └── httpd_spec.rb
│   └── spec_helper.rb
└── Vagrantfile
```

As seen in the preceding tree structure, we've chosen to write tests at a playbook level (`example1`). In this case, if we have three nodes that run the `example1` playbook, these tests in `httpd_spec.rb` will run against all three of them, provided the information of the three nodes is specified in the `Rakefile`.

Analyzing the Rakefile and running tests

Now, let's look at the `rake` tasks available and then run the tests using `rake serverspec`:

```
$ rake -T
rake serverspec        # Run serverspec on all hosts
rake serverspec:host1  # Run serverspec to host1
$ rake serverspec
------------------------------------
Running ["example1"] serverspec tests on - host1
hostname/ip - host1
/System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/bin/ruby -S rspec
C HOST --- 192.168.33.10
.F...

Failures:

  1) Service "httpd" should be enabled
     On host `192.168.33.10`
     Failure/Error: it { should be_enabled   }
       sudo chkconfig --list httpd | grep 3:on
       expected Service "httpd" to be enabled
     # ./spec/example1/httpd_spec.rb:8:in `block (2 levels) in <top (required)>'

Finished in 0.09258 seconds
5 examples, 1 failure

Failed examples:

rspec ./spec/example1/httpd_spec.rb:8 # Service "httpd" should be enabled
/System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/bin/ruby -S rspec failed
```

We see that out of five tests, the test that checks to whether the `httpd` service is enabled has failed. So let's fix that, as shown in the following command lines:

```
$ git diff playbooks/example1.yml

   - name: Check httpd service
-      service: name=httpd state=started
+      service: name=httpd state=started enabled=yes
     sudo: yes
```

Now, let's run `vagrant provision` followed by the Serverspec test.

```
$ vagrant provision
[default] Running provisioner: ansible...

PLAY [host1] *********************************************************************

TASK: [Install httpd package] ***************************************************
ok: [192.168.33.10]

TASK: [Check httpd service] *****************************************************
changed: [192.168.33.10]

PLAY RECAP **********************************************************************
192.168.33.10                : ok=2    changed=1    unreachable=0    failed=0

$ rake serverspec
-------------------------------------
Running ["example1"] serverspec tests on - host1
hostname/ip - host1
/System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/bin/ruby -S rspec
C HOST --- 192.168.33.10
.....

Finished in 0.09627 seconds
5 examples, 0 failures
```

As you can see, this is a powerful way to test the end state of your system.

> You can check out the documentation on the Serverspec site at `http://serverspec.org/` for more information.

Running playbook_tester

Tying all this together, we've a very simple `playbook_tester` that spawns the Vagrant machine, deploys the playbook, and finally runs tests on them. The following screenshot shows us how to run `playbook_tester`:

```
$ cat playbook_tester
#!/bin/bash

if [[ $# -eq 0 ]]; then
  echo "Usage: ./playbook_tester <name of playbook> "
  exit
fi

echo "Testing playbook $1"

cd $1
if [[ `vagrant status | grep running` ]] ; then
  vagrant provision
else
  vagrant up
fi
rake serverspec
```

Let's run `playbook_tester` on the example playbook.

```
$ ./playbook_tester example1
Testing playbook example1
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'devbase'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2222 (adapter 1)
[default] -- 80 => 8080 (adapter 1)
[default] Booting VM...
[default] Waiting for machine to boot. This may take a few minutes...
[default] Machine booted and ready!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
[default] Running provisioner: ansible...

PLAY [host1] ****************************************************************

TASK: [Install httpd package] **********************************************
changed: [192.168.33.10]

TASK: [Check httpd service] ************************************************
changed: [192.168.33.10]

PLAY RECAP *****************************************************************
192.168.33.10              : ok=2    changed=2    unreachable=0    failed=0


-------------------------------------
Running ["example1"] serverspec tests on - host1
hostname/ip - host1
/System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/bin/ruby -S rspec
C HOST --- 192.168.33.10
.....

Finished in 0.09085 seconds
5 examples, 0 failures
```

You can run exactly the same kind of testing in your environment or you're free to pick one of the options we've shown. This can also be integrated into your CI environment. Instead of VirtualBox, Vagrant can spawn a machine in the cloud or your environment and run functional tests for each playbook.

Handling environments

So far, we've seen how you can develop playbooks and test them. The final aspect is how to release playbooks into Production. In most cases, you will have multiple environments to deal with before the playbook is released into Production. This is similar to software that your developers have written. The progress can be shown as follows:

When you write your playbooks and set up roles (we'll discuss roles in the next chapter), we strongly recommend that you keep in mind the notion of the environments right from the start. It might be worthwhile to talk to your software and operations teams' to figure out exactly how many environments your setup has to cater to. We'll list down a couple of approaches with examples that you can follow in your environment.

Code based on Git branch

Let's assume you have four environments to take care of, which are as follows:

- Development (local)
- UAT
- Stage
- Production

In the Git branch-based method, you will have one environment per branch. You will always make changes to Development first, and then promote those changes to UAT (merge or cherry-pick, and tag commits in Git), Stage, and Production. In this approach, you will hold one single inventory file, one set of variable files, and finally, a bunch of folders dedicated to roles and playbooks per branch. Let's look at the next approach.

A single stable branch with multiple folders

In this approach, you will always maintain the `dev` and `master` branches. The initial code is committed to the `dev` branch, and once stable, you will promote it to the `master` branch (We saw this model at the beginning of this chapter.). The same roles and playbooks that exist in the `master` branch will run across all environments. On the other hand, you will have separate folders for each of your environments. Let's look at an example. We'll show how you can have a separate configuration and an inventory for two environments: `stage` and `production`. You can extend it for your scenario.

Let's first look at the playbook that will run across these multiple environments.

```
$ cat playbook.yml
---
- hosts: web
  remote_user: vagrant

  tasks:
    - name: Print environment specific url
      debug: msg={{ url }}

    - name: Print mail server
      debug: msg={{ mail_server }}

- hosts: db
  remote_user: vagrant

  tasks:
    - name: Print environment specific db password
      debug:  msg={{ db_password }}

    - name: Print mail server
      debug: msg={{ mail_server }}
```

As you can see, there are two sets of tasks in this playbook:

- Tasks that run against `db` servers
- Tasks that run against `web` servers

There is also an extra task to print the `mail_server` task that is common to all servers in a particular environment.

Let's look at the inventory for `stage` and `production` environments as follows:

```
$ cat stage/inventory
[web]
192.168.33.32

[db]
192.168.33.33

[servers:children]
db
web
$ cat production/inventory
[web]
192.168.33.30

[db]
192.168.33.31

[servers:children]
db
web
```

As you can see, we have one unique machine each (IP) for the `db` and `web` sections, respectively. Further, we have a different set of machines for `stage` and `production` environments. The additional section, `[servers:children]`, allows you to create a group of groups. In this case, the `servers` group consists of all the servers in the `db` and `web` groups. This would mean that any variables that are defined in the `servers` section will apply to both the `db` and `web` groups, unless they're overridden in the individual sections, respectively.

The next interesting part would be to look at variable values for each of the environments and see how they are separated out in each environment.

Let's look at the stage variables, as shown in the following screenshot:

```
$ tree stage/group_vars/
stage/group_vars/
├── db
├── servers
└── web

0 directories, 3 files
$ cat stage/group_vars/db
---
db_password: 'stage'
$ cat stage/group_vars/web
---
url : 'http://stage.app.com'
$ cat stage/group_vars/servers
---
mail_server: 'smtp.stage.net'
```

As you can see, we are revisiting the concept of `group_vars` with a much better example. These are variables that can be applied to an entire group. In this case, we have the following three variable files:

- `db`: This defines `db_password` for `stage`. (We've obviously simplified the command to showcase how this variable can be used. You can add whatever variables are fit for your environment.)
- `servers`: This defines `mail_server`, which is common to all servers in `stage`.
- `web`: This defines `url` for `stage`.

Similarly, let's look at the variables in the `production` environment directory.

```
$ tree production/group_vars/
production/group_vars/
├── db
├── servers
└── web

0 directories, 3 files
$ cat production/group_vars/db
---
db_password: "production"
$ cat production/group_vars/web
---
url: "http://production.app.com"
$ cat production/group_vars/servers
---
mail_server: 'smtp.prod.net'
```

The variables are structured in a similar manner. No surprises here. Now, let's run the playbook and see the effect it has on each of the environments. We'll first run it on stage, as shown in the following screenshot:

```
$ ansible-playbook -i stage/inventory  playbook.yml

PLAY [web] ***********************************************************

GATHERING FACTS *****************************************************
ok: [192.168.33.32]

TASK: [Print environment specific url] ******************************
ok: [192.168.33.32] => {
    "msg": "http://stage.app.com"
}

TASK: [Print mail server] ******************************************
ok: [192.168.33.32] => {
    "msg": "smtp.stage.net"
}

PLAY [db] ***********************************************************

GATHERING FACTS *****************************************************
ok: [192.168.33.33]

TASK: [Print environment specific db password] *********************
ok: [192.168.33.33] => {
    "msg": "stage"
}

TASK: [Print mail server] ******************************************
ok: [192.168.33.33] => {
    "msg": "smtp.stage.net"
}

PLAY RECAP *********************************************************
192.168.33.32              : ok=3    changed=0    unreachable=0    failed=0
192.168.33.33              : ok=3    changed=0    unreachable=0    failed=0
```

You can see that the Ansible run picked up all the relevant variables defined for the `stage` environment. Observe the `Print mail server` task, which prints out the `mail_server` parameter that was defined in the `servers` section. As mentioned, it's common for both the `db` and `web` servers. Similarly, let's run the same playbook in `production`, as follows:

```
$ ansible-playbook -i production/inventory  playbook.yml

PLAY [web] *********************************************************************

GATHERING FACTS ***************************************************************
ok: [192.168.33.30]

TASK: [Print environment specific url] ****************************************
ok: [192.168.33.30] => {
    "msg": "http://production.app.com"
}

TASK: [Print mail server] *****************************************************
ok: [192.168.33.30] => {
    "msg": "smtp.prod.net"
}

PLAY [db] *********************************************************************

GATHERING FACTS ***************************************************************
ok: [192.168.33.31]

TASK: [Print environment specific db password] ********************************
ok: [192.168.33.31] => {
    "msg": "production"
}

TASK: [Print mail server] *****************************************************
ok: [192.168.33.31] => {
    "msg": "smtp.prod.net"
}

PLAY RECAP ********************************************************************
192.168.33.30                  : ok=3    changed=0    unreachable=0    failed=0
192.168.33.31                  : ok=3    changed=0    unreachable=0    failed=0
```

The results are as expected. If you're using the approach to have a stable `master` branch for multiple environments, it's best to use an amalgamation of environment-specific directories, `group_vars`, and inventory groups to tackle the scenario.

Summary

With this, we conclude this chapter where you learned how to use Git with Ansible, Vagrant to develop playbooks, and finally, how to test your playbooks. We hope that you'll adopt some of the development and testing practices that we've showcased for your environment. We have the following few factors for you to think about before you proceed to the next chapter:

- Figure out how any of your development teams work and the path that their code takes to Production.

- Think about how you can start using Vagrant with Ansible to set up your playbook development environment and how you can make your entire team start using it.

- Think of what approach you would use to handle environments. How many environments do you have?

In the next chapter, we will look at various other Ansible features that you would need before taking the Ansible scripts to Production. We'll also look at how you can model the infrastructure using roles and connect multiple playbooks using the `include` feature.

3

Taking Ansible to Production

"More practice, more success"

- Ranjib Dey, an ex-colleague and a fine exponent of DevOps.

So far, we've seen how to write basic Ansible playbooks, options associated with the playbooks, practices to develop playbooks using Vagrant, and test them at the end of it. We've now got a framework for you and your team to learn and start developing Ansible playbooks. Consider this similar to learning how to drive a car from your driving school instructor. You start by learning how to control the car with the steering wheel, then you slowly begin to control the brakes, and finally, start maneuvering the gears, and hence, the speed of your car. Once you've done this over a period of time, with more and more practice on different kinds of roads (such as flat, hilly, muddy, pot-holed, and so on) and by driving different cars, you gain expertise, fluency, speed, and basically, enjoy the overall experience. From this chapter onwards, we will up the gear by digging deeper into Ansible and urge you to practice and try out more examples to get comfortable with it.

You must be wondering why the chapter is named the way it is. The reason for this is the fact that with what we've learned so far, we've not yet reached a stage where you can deploy the playbooks in Production, especially in complex situations. Complex situations include those where you have to interact with several (hundred or thousand) machines where each group of machines is dependent on another group or groups of machines. These groups may be dependent on each other for all or some transactions, to perform secure complex data backups and replications with master and slaves. In addition, there are several interesting and rather compelling features of Ansible that we've not yet looked at. In this chapter, we will cover all of them with examples. Our aim is that, by end of this chapter, you should have a clear idea of how to write playbooks that can be deployed in Production from a configuration management perspective. The following chapters will add to what we've learned to enhance the experience of using Ansible.

In this chapter, we will cover the following list of topics:

- The `local_action` feature
- Conditionals
- Loops
- The `include` feature
- Handlers
- Roles
- Templates and filters
- Security management

Working with the local_action feature

The `local_action` feature of Ansible is a powerful one, especially when we think of **Orchestration**. You will see `local_action` being used in *Chapter 6*, *Provisioning*, and *Chapter 7*, *Deployment and Orchestration*, but we felt we should definitely introduce it much earlier. This feature allows you to run certain tasks locally on the machine that runs Ansible.

Consider the following situations:

- Spawning a new machine or creating a JIRA ticket
- Managing your command center(s) in terms of installing packages and setting up configurations (Remember how we defined command centers in *Chapter 1*, *Getting Started with Ansible*, as machines from where you could run Ansible across your infrastructure?)
- Calling a load balancer API to disable a certain web server entry from the load balancer

These are tasks that can be run on the same machine that runs the `ansible-playbook` command rather than logging in to a remote box and running these commands.

Let's look at an example. Suppose you want to run a shell module on your local system where you are running your Ansible playbook. The `local_action` option comes into the picture in such situations. If you pass the module name and the module argument to `local_action`, it will run that module locally. Let's see how this option works with the shell module. Consider the following screenshot that shows the output of the `local_action` option:

```
$ cat playbooks/example1.yml
- hosts: web001
  remote_user: ec2-user
  tasks:

    - name: check running processes on remote system
      shell: ps
      register: remote_processes

    - name: remote running processes
      debug: msg="{{ remote_processes.stdout }}"

    - name: check running processes on local system
      local_action: shell ps
      register: local_processes

    - name: local running processes
      debug: msg="{{ local_processes.stdout }}"
```

In the preceding screenshot, the first and the third task run the `ps` command with the difference that the first task runs remotely and the third runs locally. The second and fourth tasks print the output. Let's execute the playbook as shown in the following screenshot:

```
$ ansible-playbook playbooks/example1.yml -i hosts

PLAY [web001] *********************************************************

GATHERING FACTS ******************************************************
ok: [web001]

TASK: [check running processes on remote system] ********************
changed: [web001]

TASK: [remote running processes] ************************************
ok: [web001] => {
    "msg": "  PID TTY          TIME CMD\n 3488 pts/0    00:00:00 sh\n 3499 pts/0    00:00:00 python\n 3500 pts/0    00:00:00 ps"
}

TASK: [check running processes on local system] ********************
changed: [web001]

TASK: [local running processes] ************************************
ok: [web001] => {
    "msg": "  PID TTY          TIME CMD\n 2018 ttys000    0:00.36 -bash\n 9689 ttys000    0:00.27 /usr/bin/python /usr/local/bin/ansible-play
book playbooks/example1.yml -i hosts\n 9704 ttys000    0:00.00 /bin/sh -c LC_CTYPE=en_US.UTF-8 LANG=en_US.UTF-8 /usr/bin/python /Users/ramesh/
.ansible/tmp/ansible-tmp-1405616859.28-125238475500898/command; rm -rf /Users/ramesh/.ansible/tmp/ansible-tmp-1405616859.28-125238475500898/ >
/dev/null 2>&1\n 9705 ttys000    0:00.07 /usr/bin/python /Users/ramesh/.ansible/tmp/ansible-tmp-1405616859.28-125238475500898/command\n  542 t
tys001    0:01.07 -bash\n 6451 ttys002    0:00.02 -bash\n  713 ttys003    0:00.00 -bash\n  716 ttys003    0:00.17 python"
}

PLAY RECAP *********************************************************
web001                     : ok=5    changed=2    unreachable=0    failed=0
```

The behavior is as expected. The first task runs remotely on the actual node (in this case, the web001 machine) and the third task runs locally on the command center. You can run any module with local_action, and Ansible will make sure that the module is run locally on the box where the ansible-playbook command is run. Another simple example you can (and should!) try is running two tasks:

- uname on the remote machine (web001 in the preceding case)
- uname on the local machine but with local_action enabled

This will crystallize the idea of local_action further.

The following is another real-world use case where local_action is used, but this time with the uri module. This use case is shown as follows:

```
$ cat playbooks/example1.yml
- hosts: web001
  remote_user: ec2-user
  tasks:

    - name: GET qoute of the day
      local_action: uri url=http://api.theysaidso.com/qod.json
      register: qod

    - name: Print quote of the day
      debug: msg="{{ qod.json.contents.quote }}"
```

In the preceding playbook, we are using the uri module with local_action. In this case, there is no need to run the uri module remotely. We register the output from the first task in qod, and with the second task, we print the quote of the day that we fetched using the uri module. The output we received from the uri module is in the JSON format; you can verify the output by running the playbook in the debug mode. Do you remember which option you need to invoke to run in the debug mode? Pat yourself on the back if you said -vv, -vvv, or -vvvv. The qod variable is a JSON variable and has the following keys: json, contents, and quote. You can invoke any of these methods. We've invoked the json.contents.quote method in the preceding screenshot.

> Let's briefly step aside and look at the `uri` module, as you will find it useful and will use it extensively if you're dealing with web services as part of your automation. The `uri` module is used to interact with web services using HTTP and HTTPS, and it supports Digest, Basic, and WSSE HTTP authentication mechanisms.
>
> As the `uri` module uses Python's `httplib2` library to interact with web services, make sure you have the `httplib2` library installed before you start using the `uri` module. You can install the `httplib2` library using `pip install httplib2`. By default, the `uri` module will use the HTTP `GET` method. However, depending on the situation, you can use other HTTP methods, using the `method` option of the `uri` module, which supports the `GET`, `POST`, `PUT`, `HEAD`, `DELETE`, `OPTIONS`, and `PATCH` methods.

Let's now run the preceding example playbook with `uri` and `local_action` together as follows:

```
$ ansible-playbook -i hosts playbooks/example1.yml

PLAY [web001] *************************************************************

GATHERING FACTS **********************************************************
ok: [web001]

TASK: [GET qoute of the day] *********************************************
ok: [web001]

TASK: [Print quote of the day] *******************************************
ok: [web001] => {
    "msg": "I still find each day too short for all the books I want to read... and purchase. -John Burrough and I. XD"
}

PLAY RECAP ***************************************************************
web001                     : ok=3    changed=0    unreachable=0    failed=0
```

As you can see, the `uri` module fetched the quote of the day from `api.theysaidso.com` and printed it.

Working with conditionals

Till now, we saw how playbooks work and how tasks are executed. We also saw that Ansible executes all these tasks sequentially. However, this would not help you while writing an advanced playbook that contains tens of tasks and you have to execute only a subset of these tasks. For example, let's say you have a playbook that will install Apache HTTPD Server on the remote host. Now, the Apache HTTPD Server has a different package name for a Debian-based operating system, and it's called `apache2`; for a Red-Hat-based operating system, it's called `httpd`.

Having two tasks, one for the `httpd` package (for Red-Hat-based systems) and the other for the `apache2` package (for Debian-based systems) in a playbook, will make Ansible install both packages, and this execution will fail, as `apache2` will not be available if you're installing on a Red-Hat-based operating system. To overcome such problems, Ansible provides conditional statements that help run a task only when a specified condition is met. In this case, we do something similar to the following pseudocode:

```
If os = "redhat system"
  Install httpd
Else if os = "debian system"
  Install apache2
End
```

While installing `httpd` on a Red-Hat-based operating system, we first check whether the remote system is running a Red-Hat-based operating system, and if yes, we then install the `httpd` package; otherwise, we skip the task. Without wasting your time, let's dive into an example playbook:

```
$ cat playbooks/example1.yml
---
- hosts: web001
  remote_user: ec2-user
  tasks:
   - name: Install httpd package
     yum: name=httpd state=latest
     sudo: yes
     when: ansible_os_family == "Redhat"

   - name: Install apache2 package
     apt: name=apache2 state=latest
     sudo: yes
     when: ansible_os_family == "Debian"
```

In the preceding playbook, we first check whether the value of `ansible_os_family` is Red Hat, and if yes, then install the `httpd` service with the latest version.

> Note that `ansible_os_family` is a fact that is collected by Ansible whenever you run a playbook, provided that you have not disabled this feature using `gather_facts: False`.

The second task in the playbook will install the latest `apache2` package if the `ansible_os_family` is Debian. If the `when` condition doesn't match a task, then it will simply skip that task and move forward. You can also use user-defined variables or a registered variable instead of Ansible facts.

Let's look at another example where we use conditionals.

```
$ cat playbooks/example1.yml
---
- hosts: web001
  remote_user: ec2-user
  tasks:
    - name: Testing user sudo privilege
      command: /usr/bin/sudo -v
      register: sudo_response
      ignore_errors: yes

    - name: Stop further if User doesn't have sudo privilege
      fail: msg="User doesn't have sudo privilege."
      when: sudo_response.rc == 1
```

In the preceding playbook, we check whether the user has `sudo` permissions on the remote host. We've seen this scenario quite often when the remote user requires the `sudo` permission to execute certain tasks, for example, installing a package

The first task will run a command, `/usr/bin/sudo -v`, on the remote host and store its output in a response variable, in this case, the `sudo_response` variable. We use the `register` feature to capture the output (that is, `stdout`) of the `sudo` command. Apart from `stdout`, the `register` variable will also capture `stderr` (if any), the **return code** (**rc**) of the command, the command itself, the start and end time of the command being executed, and so on.

In the second task, we check whether the rc of the command executed in the previous task was 1. We use the `fail` module to exit the Ansible run with a message if the return code was 1. If the `when` condition doesn't match, (that is, the return code is not 1) Ansible will simply skip that task. Notice that we are using `sudo_response.rc` instead of `sudo_response` in the `when` condition. This is because Ansible stores the output of any module in Python's dictionary format (that is, the key-value pair) so that you can traverse over various keys easily. Let's see how this playbook works:

```
$ ansible-playbook -i hosts playbooks/example1.yml

PLAY [web001] ******************************************************************

GATHERING FACTS ***************************************************************
ok: [web001]

TASK: [Testing user sudo privilege] *******************************************
changed: [web001]

TASK: [Stop further if User doesn't have sudo privilege] **********************
skipping: [web001]

PLAY RECAP ********************************************************************
web001                     : ok=2    changed=1    unreachable=0    failed=0
```

In the preceding example, the first task ran but the second task was skipped by Ansible, as the `when` condition for that task failed because the rc of the `sudo` command in the previous task was `0`.

Likewise, you can have negative conditions as well, where you can check whether the output of a command is not equal to the expected output. Let's see an example task for this in the following screenshot:

```
- name: Stop further if User doesn't have sudo privilege
  fail: msg="User doesn't have sudo privilege."
  when: sudo_response.rc != 0
```

In the preceding task, instead of checking whether the response code is `1`, we check whether it is nonzero, and here, `!=` means `not equal to`.

> Apart from `!=`, Ansible also supports the `>`, `<`, `<=`, `>=`, and `==` operators with the `when` condition.

The preceding operator will match the entire content of the variable, but what if you just want to check whether a particular character or a string is present in a variable? To perform such kinds of checks, Ansible provides the `in` and `not` operators, as shown in the following screenshot:

```
- name: Querying rpm list for httpd package
  shell: rpm -qa | grep httpd
  register: httpd_rpm

- name: Check if httpd rpm is installed on the remote host
  debug: msg="httpd is installed on the remote host"
  when: "'httpd-2.2.27-1.2.amzn1.x86_64' in httpd_rpm.stdout"

- name: Check if httpd rpm is installed on the remote host
  debug: msg="httpd is not installed on the remote host"
  when: not 'httpd-2.2.27-1.2.amzn1.x86_64' in httpd_rpm.stdout
```

In the preceding example, we first get a list of all the RPM files installed, use the `grep` command for a specific version of the `httpd` package, and store the output in a variable. We then check whether the string, `httpd-2.2.27-1.2.amzn1.x86_64`, is present in the `httpd_rpm` variable using the `when` condition. The third task in the preceding example is the opposite of the second task; it will check whether the string, `httpd-2.2.27-1.2.amzn1.x86_64`, is not present in the `httpd_rpm` variable. You can also match multiple conditions using the `and` and `or` operators. The `and` operator will make sure that all conditions are matched before executing this task, where the `or` operator will make sure that at least one of the conditions match.

Consider another example given in the following screenshot:

```
- name: Querying rpm list for httpd package
  shell: rpm -qa | grep httpd
  register: httpd_rpm

- name: Check if httpd rpm is installed on the remote host
  debug: msg="httpd is installed on the remote host"
  when: "'httpd-2.2.27-1.2.amzn1.x86_64' in httpd_rpm.stdout and 'httpd-tools-2.2.27-1.2.amzn1.x86_64' in httpd_rpm.stdout"

- name: Check if httpd rpm is installed on the remote host
  debug: msg="httpd is not installed on the remote host"
  when: not 'httpd-2.2.27-1.2.amzn1.x86_64' in httpd_rpm.stdout or not 'httpd-tools-2.2.27-1.2.amzn1.x86_64' in httpd_rpm.stdout
```

In the preceding screenshot, the first task remains the same. In the second task, we check whether the strings, `httpd-2.2.27-1.2.amzn1.x86_64` and `httpd-tools-2.2.27-1.2.amzn1.x86_64`, are present in the `httpd_rpm` variable. Likewise, you can have multiple `and` operators, and Ansible will make sure that all of them are matched before executing the task.

In the third task, using the `or` operator, we check whether neither of the two strings are present in the `httpd_rpm` variable. Ansible will skip this task if either of the conditions matches.

> You can also combine both the `or` and `and` operators together, but we will leave this exercise for you to try.

Apart from string matching, you can also check whether a variable exists. This type of validation will be useful when you want to check whether a variable was assigned some value or not. You can even execute a task based on the Boolean value of a variable.

Consider the example given in the following screenshot:

```
- name: Rsync the entire disk
  shell: /usr/bin/rsync -ra / /backup/{{ inventory_hostname }} --exclude='/proc' --exclude='/sys' --exclude='/backup'
  sudo: yes
  when: backup
```

The preceding task is an example of backing up your entire disk using `rsync`. The `rsync` command is a Linux command to copy files. We have used the `when` condition here to check whether the `backup` variable is set to `true`; that is, Ansible will check the Boolean value of the variable and run the task only if it is set to `true` or `1`. The value for the `backup` variable can either be set in the variable file, or you can ask the user to pass a `backup` variable using the `--extra-vars` option or prompt the user for it.

Ansible also provides a way to check whether a variable was defined or not, which is shown in the following screenshot:

```
- name: Checking backup path
  pause: prompt="Please provide a path for backup"
  register: backup_path
  when: backup_path is not defined

- name: Rsync the entire disk
  shell: /usr/bin/rsync -ra / {{ backup_path.user_input }} --exclude='/proc' --exclude='/sys' --exclude='/backup'
  sudo: yes
  when:  backup
```

In the preceding example, we first check whether the `backup_path` variable is defined, and if it is not, we prompt the user for the backup path. In the second task, we use the `backup_path` variable with `rsync` to back up the entire disk on that path. Likewise, you can also apply a condition to check the existence of a variable. Consider the example in the following screenshot:

```
- name: Rsync the entire disk
  shell: /usr/bin/rsync -ra / {{ backup_path }} --exclude='/proc' --exclude='/sys' --exclude='/backup'
  sudo: yes
  when:  backup and backup_path is defined
```

In the preceding case, we check whether the `backup` variable is set to `true` and the `backup_path` variable exists (it doesn't matter what value it's set to).

Working with loops

Have you ever wondered how life would be if you had to install several packages on your remote systems manually? We saw how Ansible helped us in such a situation by automating the package installation. However, it would be still a nightmare if you have to write the same task multiple times with different package names. To overcome such situations and to make your life easier, Ansible provides you with loops.

Standard loops

Using standard loops, you can pass a list of packages to install and Ansible will run that task for all packages listed. Confused? Let's now go through an example playbook.

```
$ cat playbooks/example1.yml
---
- hosts: web001
  remote_user: ec2-user
  tasks:
    - name: Installing packages
      yum: name={{ item }} state=latest
      with_items:
          - httpd
          - mysql
          - mysql-server
      sudo: yes
```

In the preceding playbook, Ansible will run the `Installing packages` task multiple times (in this case, three times): once for each of the packages that are defined under the `with_items` construct. Each of the packages is exposed as `item`, which is a default variable that Ansible creates. Ansible then assigns a package name to `item`, based on the iteration it is currently part of. So, for the first iteration, `item` is set to `httpd`; in the second iteration, it is set to `mysql`; and in the final iteration, it is set to `mysql-server`. Let's run this playbook and see how this works in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/example1.yml

PLAY [web001] ******************************************************************

GATHERING FACTS ***************************************************************
ok: [web001]

TASK: [Installing packages] ***************************************************
changed: [web001] => (item=httpd,mysql,mysql-server)

PLAY RECAP ********************************************************************
web001                     : ok=2    changed=1    unreachable=0    failed=0
```

Ansible ran the task once, but it looped through the package installation using `yum` for all three packages that we listed. This was the simplest way of looping in Ansible; you can also pass a dictionary to `with_items` instead of a string, as shown in the following screenshot:

```
- name: Installing packages
  yum: name={{ item.name }} state={{ item.state }}
  with_items:
      - {name: 'httpd', state: 'latest'}
      - {name: 'mysql', state: 'absent'}
      - {name: 'mysql-server', state: 'absent'}
  sudo: yes
```

Ansible will assign all dictionaries to the `item` variable one by one. This type of loop will be useful when you have multiple arguments for a module; for example, with the package name, you would also want to tell Ansible the expected state of that package, that is, whether it should be present or absent.

Nested Loops

Nested loops are useful when you have to perform multiple operations on the same resource. For example, you want to allow access to multiple databases to a MySQL user(s). To perform this task, you might first think of writing multiple tasks, one for each group access, but it would be more difficult when you have more than one user. To make such tasks easy, nested loops are used. Let's see an example playbook:

```
$ cat playbooks/example1.yml
---
- hosts: web001
  remote_user: ec2-user
  tasks:
    - name: give users access to multiple databases
      mysql_user: name={{ item[0] }} priv={{ item[1] }}.*:ALL append_privs=yes password=foo login_user=root login_password=root
      with_nested:
        - [ 'alice', 'bob' ]
        - [ 'clientdb', 'employeedb', 'providerdb' ]
```

In the preceding playbook, we use the `mysql_user` module to give permissions to a specific database. We use nested loop (using the `with_nested` parameter) with two lists: the first is the user list, and the other is the database list. Ansible will run the database list for each user in the first list; that is, first, Ansible will run the `mysql_user` module for the user, `alice`, giving permissions to all three databases one by one, and then, it will run for the user, `bob`. You can have multiple nested lists, and Ansible will loop over all of them one by one. Let's see how this works:

```
$ ansible-playbook -i hosts playbooks/example1.yml

PLAY [web001] ********************************************************

GATHERING FACTS *****************************************************
ok: [web001]

TASK: [give users access to multiple databases] ********************
changed: [web001] => (item=['alice', 'clientdb'])
changed: [web001] => (item=['alice', 'employeedb'])
changed: [web001] => (item=['alice', 'providerdb'])
changed: [web001] => (item=['bob', 'clientdb'])
changed: [web001] => (item=['bob', 'employeedb'])
changed: [web001] => (item=['bob', 'providerdb'])

PLAY RECAP *********************************************************
web001                     : ok=2    changed=1    unreachable=0    failed=0
```

As expected, Ansible ran the database list for both users one by one; likewise, you can have multiple lists, and Ansible will loop over all of them.

Looping over subelements

Till now, we saw how loops work with static data; that is, if you want to give permission to access a database to multiple users with same set of database. You cannot specify different sets of databases to different users using the preceding loops. To deal with such cases, you can use a dictionary to loop over subelements. Using such loops, you can specify a set of databases per user.

Consider the example in the following screenshot:

```
$ cat playbooks/example1.yml
---
- hosts: web001
  remote_user: ec2-user
  vars:
    users:
      - name: alice
        database:
          - clientdb
          - employeedb
          - providerdb
      - name: bob
        database:
          - clientdb

  tasks:
    - name: give users access to multiple databases
      mysql_user: name={{ item.0.name }} priv={{ item.1 }}.*:ALL append_privs=yes password=foo login_user=root login_password=root
      with_subelements:
        - users
        - database
```

In the preceding playbook, we created a dictionary variable, which consists of username and database names. Instead of adding user data in the playbook itself, you can also add it to a separate variable file and include it in your playbook. Ansible will then loop over this dictionary using the `item` variable. Ansible will assign numeric values to the keys you provided using `with_subelements`, starting from `0`.

In the preceding example playbook, you can access users using `item.0` and the database using `item.1`. In the preceding dictionary, the name is a key-value pair, which is the reason we are accessing it using `item.0.name`, whereas the database is a list, because of which we can directly access it using `item.1`. Let's run this playbook and see how it works. The output is shown in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/example1.yml

PLAY [web001] ************************************************************

GATHERING FACTS *********************************************************
ok: [web001]

TASK: [give users access to multiple databases] ************************
ok: [web001] => (item=({'name': 'alice'}, 'clientdb'))
ok: [web001] => (item=({'name': 'alice'}, 'employeedb'))
ok: [web001] => (item=({'name': 'alice'}, 'providerdb'))
ok: [web001] => (item=({'name': 'bob'}, 'clientdb'))

PLAY RECAP **************************************************************
web001                     : ok=2    changed=0    unreachable=0    failed=0
```

Working with include

In *Chapter 1*, *Getting Started with Ansible*, we've seen a quick example of `include` in the context of variables. However, it's not just variables that we include from other files; it can also be common tasks or handlers. This is a feature in Ansible to reduce duplicity while writing tasks. This also allows us to have smaller playbooks by including reusable code in separate tasks using **Don't Repeat Yourself** (**DRY**). Yes, this is exactly what you do in programming as well. Let's consider the following scenario.

For example, imagine that you have the `httpd` service and you want to have local monitoring that makes sure that the `httpd` service is always up. The `monit` package from Linux is popularly used to automatically start services in the case where services stop, and thus, this package can be used to solve the case where `httpd` should always be up. Let's consider a situation where `monit` is being used to monitor a custom service (call it SVC A) that you have written, and to deploy a change, you need to stop a service, update the package, and then start the service. So, one of the prerequisites here will be to stop `monit` before stopping SVC A. Let's say, you've now stopped `monit`, updated the package associated with SVC A, started up SVC A, and you want to start `monit` again. There might be multiple such scenarios and playbooks where you want to manage services. The `include` feature works very well in such cases. Let's create a `manage_service.yml` task that can be included in the main playbook.

```
$ cat playbooks/tasks/manage_service.yml
---
- name: "{{ service_state}} {{ service_name }}"
  sudo: yes
  service: name={{ service_name }} state={{ service_state }}
```

As you can see in the preceding screenshot, there are two variables that you need to provide in order to use the service:

- `service_name`
- `service_state`

Each of these can either be variables in the playbook under the `vars` section, as shown in the following screenshot:

```
$ cat playbooks/example3.yml
---
- hosts: host1
  gather_facts: False
  remote_user: vagrant

  vars:
    - service_name: "monit"
    - service_state: "stopped"

  tasks:
    - include: tasks/manage_service.yml
```

Observe that we've used `include` to include `manage_service.yml`. The variables can also be hardcoded as shown in the following screenshot:

```
  tasks:
    - include: tasks/manage_service.yml service_name="monit" service_state="stopped"
```

The overall structure for the playbook currently looks like the following:

```
$tree
├── Vagrantfile
├── inventory
└── playbooks
    ├── example3.yml
    └── tasks
        └── manage_service.yml


2 directories, 4 files
```

Now, let's run the playbook. The result is as expected in both cases. Note that the task name also can be parameterized.

```
$ ansible-playbook -i inventory playbooks/example3.yml --private-key=~/.ssh/ansible_key

PLAY [host1] ********************************************************************

TASK: [stopped monit] **********************************************************
changed: [192.168.33.10]

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

We'll look at handlers next and revisit `include` when we look at roles.

Working with handlers

In many situations, you will have a task or a group of tasks that change certain resources on the remote machines. Ansible recognizes these changes. However, for these changes to be truly effective, they need to trigger other events. For example, when package or application configuration files change, you will need to restart or reload the associated service. Normally, this won't happen as the associated service is already running, and when the Ansible task runs for that service, it figures out that the service is already running, and hence, there won't be any change. However, you will require a way in which a restart event can be triggered for the associated service. This is done via the `notify` action, and tasks that are called by this `notify` action are called handlers.

Every handler task will run at the end of the playbook if previously notified, for example, you changed your `httpd` server config multiple times and you want to restart the `httpd` service so that the changes are applied. Now, restarting `httpd` every single time on a configuration change is not a good practice. To deal with such a situation, you can notify Ansible to restart the `httpd` service on every configuration change, but Ansible will make sure that no matter how many times you notify it for the `httpd` restart, it will call that task just once after all other tasks complete. Let's see how this works in the following screenshot:

```
$ cat playbooks/example1.yml
---
- hosts: web001
  remote_user: ec2-user
  tasks:
    - name: Create a virtual host
      template: src=httpd_test.conf_web dest=/etc/httpd/conf/test.conf mode=644 validate='httpd -t -f %s'
      sudo: yes
      notify:
        - restart httpd

  handlers:
    - name: restart httpd
      service: name=httpd state=restarted
      sudo: yes
```

In the preceding playbook, we have defined the `restart httpd` handler. When the virtual host is created, it notifies the handler, and the handler restarts the `httpd` service. Remember, the task itself is idempotent, and hence, the handler is notified only the first time. If the configuration pertaining to the virtual host `test.conf` file changes, then again Ansible will reinforce the intended configuration and notify the handler.

To notify the handler task, we use the `notify` parameter that tells Ansible to call the `restart httpd` task at the end of the playbook. As mentioned earlier, you can call the handler task multiple times using the `notify` parameter, but Ansible will run it just once at the end of the playbook. Let's run the playbook to see whether the behavior is as we expected it to be. The output can be seen in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/example1.yml

PLAY [web001] ************************************************************

GATHERING FACTS *********************************************************
ok: [web001]

TASK: [Create a virtual host] ******************************************
changed: [web001]

NOTIFIED: [restart httpd] **********************************************
changed: [web001]

PLAY RECAP *************************************************************
web001                     : ok=3    changed=2    unreachable=0    failed=0
```

> Handlers are triggered only when a task status is `changed`.

The results are as expected. Ansible updated the `test.conf` file and notified the handler to restart the `httpd` service. You can also call multiple handlers from the same task, as shown in the following screenshot:

```
$ cat playbooks/example1.yml
---
- hosts: web001
  remote_user: ec2-user
  tasks:
    - name: Update memcached configuration
      template: src=memcached_web dest=/etc/sysconfig/memcached mode=644
      sudo: yes
      notify:
        - restart memcached

    - name: Create a virtual host
      template: src=httpd_test.conf_web dest=/etc/httpd/conf/test.conf mode=644 validate='httpd -t -f %s'
      sudo: yes
      notify:
        - restart httpd
        - restart memcached

  handlers:
    - name: restart httpd
      service: name=httpd state=restarted
      sudo: yes

    - name: restart memcached
      service: name=memcached state=restarted
      sudo: yes
```

As discussed earlier, you can call the same handler multiple times, for example, in the preceding playbook, we call the `restart memcached` handler two times and Ansible will make sure that `memcached` is bounced only once at the end of the playbook. You can also include a handler file instead of writing individual handlers in a playbook. This is shown in the following screenshot:

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

Let's now proceed to roles.

Working with roles

When you start thinking about your infrastructure, you will soon look at the purposes each node in your infrastructure is solving and you will be able to categorize them. You will also start to abstract out information regarding nodes and start thinking at a higher level. For example, if you're running a web application, you'll be able to categorize them broadly as `db_servers`, `app_servers`, `web servers`, and `load balancers`.

If you then talk to your provisioning team, they will tell you which base packages need to be installed on each machine, either for the sake of compliance or to manage them remotely after choosing the OS distribution or for security purposes. Simple examples can be packages such as `bind`, `ntp`, `collectd`, `psacct`, and so on. Soon you will add all these packages under a category named `common` or `base`. As you dig deeper, you might find further dependencies that exist. For example, if your application is written in Java, having some version of JDK is a dependency. So, for what we've discussed so far, we have the following categories:

- `db_servers`
- `app_servers`
- `web_servers`
- `load_balancers`
- `common`
- `jdk`

We've taken a top-down approach to come up with the categories listed. Now, depending on the size of your infrastructure, you will slowly start identifying reusable components, and these can be as simple as `ntp` or `collectd`. These categories, in Ansible's terms, are called **Roles**. If you're familiar with Chef, the concept is very similar. We will look at the end-to-end deployment with roles later in the book, but for now let's consider a simpler example to start with.

Let's consider the build system. In most cases, you will expect a certain number of components to be installed on your build agents to run these builds. At the highest level, we'll start with the `build_agent` role that needs to be applied on all the machines that are running builds. Now, if you have multiple applications, for example, a Tomcat app and a Rails app to build, you will have two roles: `tomcat_build_agent_role` and `rails_build_agent_role`. Each role can have other roles as dependencies. We'll consider a Java application that requires JDK, Maven, and Ant (popular build tools for Java applications) on all the build agents. Furthermore, let's consider that the Java application interacts with Cassandra. This can be any other database but we've chosen Cassandra here for a reason; however, the same techniques apply in other similar situations. Cassandra will run on another machine so that integration tests for the code are run against that Cassandra node. Let's look at multiple ways to model this:

- Create two roles as follows:
    - The `Build Agent` role that has Ant, Maven, and JDK as task files
    - The `Cassandra` role with JDK, again as a task file, and Cassandra as another task file

- Create three roles and one task file as follows:
    - ◦ The `JDK` role
    - ◦ The `Build Agent` role with Ant and Maven as tasks and the `JDK` role as dependency
    - ◦ The `Cassandra` role with the JDK role as dependency

Clearly, the second way makes more sense, keeping in mind the DRY policy. However, let's look at how the code will look for each of the preceding cases and go over each option in more detail.

Let's create two roles with three task files. Running the `tree` command inside the `build_agent` role directory, we see the following output:

```
$ cd build_agent
$ tree
.
├── tasks
│   ├── ant.yml
│   ├── jdk.yml
│   ├── main.yml
│   └── maven.yml
├── templates
│   ├── ant.sh
│   └── maven.sh
└── vars
    └── main.yml


3 directories, 7 files
```

As you can see, we have three parallel folders: `tasks`, `vars`, and `templates`. The tasks folder utilizes the variable values in `main.yml` and the templates to make sure the desired state is achieved on the machines.

When it comes to roles, there is a default `main.yml` file that will always be executed. Using the `include` feature that we saw in the previous examples, we include other task files that will be executed in the order they are included. Let's look at the content of `main.yml` as follows:

```
$ cat tasks/main.yml
---
- include: jdk.yml
- include: ant.yml
- include: maven.yml
```

In this case, the YAML files that will be executed are `jdk.yml`, `ant.yml`, and `maven.yml`, in this order.

We have simplified, yet working, versions of the YAML files, as our focus here is on roles. Let's now look at the content of these files in the following screenshot:

```
$ cat tasks/jdk.yml
---
  - name: download jdk rpm
    get_url: url=http://{{ fileserver }}/{{ jdk6 }} dest=/tmp thirsty=no
    tags:
      - jdk

  - name: setup java jdk
    yum: name=/tmp/{{ jdk6 }} state=present
    tags:
      - jdk
$ cat tasks/ant.yml
---
  - name: create ant directory
    file: dest={{ ant_base_directory }} state=directory owner=root group=root
    tags:
     - ant

  - name: download ant
    get_url: url=http://{{ fileserver }}/{{ ant_software }} dest={{ ant_base_directory }} thirsty=no
    tags:
     - ant

  - name: untar ant
    shell: cd {{ ant_base_directory }}; tar zxvf {{ ant_software }} creates={{ ant_base_directory }}/{{ ant_directory }}
    tags:
     - ant

  - name: add ant to /etc/profile.d
    template: src=ant.sh dest=/etc/profile.d/ant.sh mode=0644 owner=root group=root
    tags:
     - ant
$ cat tasks/maven.yml
---
  - name: download maven
    get_url: url={{ maven_url }} dest={{ maven_base_directory }} thirsty=no
    tags:
      - maven

  - name: untar maven
    shell: cd /opt; tar -zxvf {{ maven_software }} creates=/opt/{{ maven_directory }}
    tags:
      - maven

  - name: add maven file to /etc/profile.d
    template: src=maven.sh dest=/etc/profile.d
    tags:
      - maven
```

You might notice that the `hosts` section is not present as it was in all our previous cases. For roles, Ansible allows us to define tasks and provide the inventory externally as we'll see shortly. This is also similar to the way Puppet and Chef manage manifests and recipes, respectively. Let's now look at the values in the variable file, `main.yml`, as shown in the following screenshot:

```
$ cat vars/main.yml
#Ant related variable configurations
ant_base_directory: /opt/
ant_software: apache-ant-1.7.1.tar.gz
ant_directory: apache-ant-1.7.1

fileserver: 172.16.1.62
jdk7: jdk-7u51-linux-x64.rpm
jdk6: jdk-6u45-linux-amd64.rpm

#maven variables
maven_url: "http://172.16.1.62/apache-maven-3.0.5-bin.tar.gz"
maven_software: apache-maven-3.0.5-bin.tar.gz
maven_directory: apache-maven-3.0.5
maven_base_directory: /opt/
```

If you're wondering how the inventory and playbooks are tied up together, it's via the `site.yml` file that exists at the same level as the `roles` folder. For any new project that has roles, it's important to have a single `site.yml` file that can have one or more included files or all the mappings right there. We'll look at how to deal with `site.yml` when we create the next role, as we'll have two roles to play with. For now, the contents of `site.yml` are as follows:

```
$ cat site.yml
- hosts: build_agents
  user: vagrant
  sudo: yes
  roles:
   - build_agent
  tags:
   - build_agent
$ cat inventory
[build_agents]
192.168.33.10
```

The `site.yml` file, which consists of a role, the inventory, and other attributes such as `tags`, `user`, and `sudo`, represents what in Ansible parlance is called a **Play**. So, we have an Ansible play that runs the `build_agent` role against a host group called `build_agents`. As you can see later, we can define multiple plays in the same `site.yml` file. For now, we'll proceed with this play. The `roles` section in `site.yml` can have one or more roles as well. Our inventory section has only one machine in this case.

To summarize, the `site.yml` file contains a play that will end up configuring the machine with the IP, `192.168.33.10`, as a build agent with the `build_agent` role. It's really important that you understand and start using the same nomenclature to express what's happening and the tool will automatically begin to appear a lot simpler. Let's execute `site.yml` via the Ansible playbook and see what happens:

```
$ ansible-playbook -i inventory  --private-key ~/.ssh/ansible_key site.yml

PLAY [build_agents] *******************************************************

GATHERING FACTS ***********************************************************
ok: [192.168.33.10]

TASK: [build_agent | download jdk rpm] ************************************
changed: [192.168.33.10]

TASK: [build_agent | setup java jdk] **************************************
changed: [192.168.33.10]

TASK: [build_agent | create ant directory] *******************************
ok: [192.168.33.10]

TASK: [build_agent | download ant] ***************************************
changed: [192.168.33.10]

TASK: [build_agent | untar ant] ******************************************
changed: [192.168.33.10]

TASK: [build_agent | add ant to /etc/profile.d] **************************
changed: [192.168.33.10]

TASK: [build_agent | download maven] *************************************
changed: [192.168.33.10]

TASK: [build_agent | untar maven] ****************************************
changed: [192.168.33.10]

TASK: [build_agent | add maven file to /etc/profile.d] *******************
changed: [192.168.33.10]

PLAY RECAP ***************************************************************
192.168.33.10                  : ok=10    changed=8    unreachable=0    failed=0
```

You can see from the preceding screenshot that every task name starts with the role name, `build_agent`. It follows the same order: first, it configures the JDK, then Ant, and finally, Maven defined in `main.yml` of the `tasks` folder. Further, we have eight changes. Only two `ok` tasks are present, because the ant directory (`/opt`) exists and the gathering facts task does not bring about any change on the box.

Before we proceed to the next role, let's look at the playbook in more detail. As we spoke so much about testing and Vagrant in the previous chapter, we'll use Vagrant to provision the machine, and followed by this, we'll test the preceding run with `serverspec` tests. (We also recommend `serverspec` tests because it's a different tool than what we've used to configure the system and also because its primary aim is to test!)

The following screenshot shows `Vagrantfile`:

```ruby
$ cat Vagrantfile
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "devbase"
  config.vm.network :forwarded_port, guest: 80, host: 8080
  config.vm.network :private_network, ip: "192.168.33.10"
  config.vm.provision "ansible" do |ansible|
    ansible.inventory_path = "inventory"
    ansible.playbook = "site.yml"
    ansible.sudo = true
    ansible.limit = 'all'
  end
end
```

The following is the result of `vagrant provision`:

```
$ vagrant provision
[default] Running provisioner: ansible...

PLAY [build_agents] *********************************************************

GATHERING FACTS ************************************************************
ok: [192.168.33.10]

TASK: [build_agent | download jdk rpm] ************************************
changed: [192.168.33.10]

TASK: [build_agent | setup java jdk] **************************************
changed: [192.168.33.10]

TASK: [build_agent | create ant directory] *******************************
ok: [192.168.33.10]

TASK: [build_agent | download ant] ***************************************
changed: [192.168.33.10]

TASK: [build_agent | untar ant] ******************************************
changed: [192.168.33.10]

TASK: [build_agent | add ant to /etc/profile.d] **************************
changed: [192.168.33.10]

TASK: [build_agent | download maven] *************************************
changed: [192.168.33.10]

TASK: [build_agent | untar maven] ****************************************
changed: [192.168.33.10]

TASK: [build_agent | add maven file to /etc/profile.d] *******************
changed: [192.168.33.10]

PLAY RECAP ***************************************************************
192.168.33.10              : ok=10    changed=8    unreachable=0    failed=0
```

Let's look at `Rakefile`, which is shown in the following screenshot:

```
$ cat Rakefile
require 'rubygems'
require 'bundler/setup'
require 'rake'
require 'rspec/core/rake_task'
#require 'json'
hosts = [
  {
    :name  => 'host1',
    :ip    => '192.168.33.10',
    :role => %w( build_agent ),
  },
]

desc "Run serverspec on all hosts"
task :serverspec => 'serverspec:all'

namespace :serverspec do
  #If you have a lot of hosts and want to run it in parallel
  #multitask :all => hosts.map {|h| 'serverspec:' + h[:name] }
  task :all => hosts.map {|h| 'serverspec:' + h[:name] }
  hosts.each do |host|
    desc "Run serverspec to #{host[:name]}"
    RSpec::Core::RakeTask.new(host[:name].to_sym) do |t|
      ENV['TARGET_HOST'] = host[:ip]
      puts "----------------------------------"
      puts "Running #{host[:role]} serverspec tests on - #{host[:name]} "
      t.pattern = "spec/#{host[:role]}/*_spec.rb"
      puts "hostname/ip - #{host[:name]}"
    end
  end
end
```

In `Rakefile`, you can see that we now have our folder structure based on roles (`:role => %w( build_agent)` and `t.pattern = spec/#{host[:role]}/*_spec.rb`) rather than playbooks, which is what we saw in the previous chapter. So, you can basically tweak the required parameters in `Rakefile` to make sure that the right abstraction (`role` in this case and previously at the playbook level) has been utilized.

It is now time to view the folder structure of our tests and the tests as follows:

```
$ tree spec/
spec/
├── build_agent
│   ├── ant_spec.rb
│   ├── jdk_spec.rb
│   └── maven_spec.rb
└── spec_helper.rb

1 directory, 4 files
$ cat spec/build_agent/*
require 'spec_helper'

describe command('which ant') do
    it  { should return_stdout '/opt/apache-ant-1.7.1/bin/ant' }
end

describe file('/opt/apache-ant-1.7.1') do
    it { should be_directory }
end
require 'spec_helper'

describe package('jdk') do
  it { should be_installed }
end

describe command('which java') do
  it { should return_stdout '/usr/bin/java' }
end
require 'spec_helper'

describe command('which mvn') do
    it { should return_stdout '/opt/apache-maven-3.0.5/bin/mvn' }
end

describe file('/opt/apache-maven-3.0.5') do
    it { should be_directory }
end
```

Let's run the tests using `rake Serverspec` as follows:

```
$ rake serverspec
--------------------------------------
Running ["build_agent"] serverspec tests on - host1
hostname/ip - host1
/System/Library/Frameworks/Ruby.framework/Versions/2.0/usr/bin/ruby -S rspec
C HOST --- 192.168.33.10
......

Finished in 0.06033 seconds
6 examples, 0 failures
$
```

On running the tests, we end up with a happy smile, as all our tests confirm that our intended state is actually now the desired state on the system.

Before we jump to the `Cassandra` role, we'd like to briefly talk about the `get_url` module that we've used in `ant.yml` and `maven.yml`, as that is the only new module that we've used. By now, we guess you should be comfortable reading tasks around file, shell, and template modules. If not, we'd recommend you to revisit the *Working with modules* section of *Chapter 1, Getting Started with Ansible*. Remember what we said at the beginning of this chapter: *more practice, more success*.

The `get_url` task (see the following example) takes a URL as a parameter from where you intend to download content, to a particular destination `dest`. It supports `http`, `https`, and `ftp`. It supports authentication as well, which we've not used in our following examples, as we've considered that we have an internal file server to download both Ant and Maven. The URLs for Ant and Maven in this case will be updated only when the versions change. The `get_url` module also gives you a `thirsty` property or a `force` property (the `thirsty` property is an alias of `force`), which is by default set to `no`. This means that if the file of that name already exists, then it won't be downloaded. If you set this property to `yes`, the behavior of the module changes and the file is downloaded even if it's already present.

The `get_url` tasks are shown as follows:

```
- name: download ant
   get_url: url=http://{{ fileserver }}/{{ ant_software }} dest={{ ant_
base_directory }} thirsty=no
   tags:
    - ant
- name: download maven
   get_url: url={{ maven_url }} dest={{ maven_base_directory }}
thirsty=no
   tags:
maven
```

Also, note that the ways in which we've used the two URL parameters in the two tasks are different. It's better to use the option we've used with `ant_software` rather than the one with `maven_software` as, even if the `fileserver` parameter changes, we change it at a global level rather than changing it at each individual value and everything again begins to work.

The Cassandra role

Let's now look at the second role, which is the `Cassandra` role. In the following example, the `Cassandra` role comes with JDK as a dependency, and hence a `jdk` task file and `cassandra` as another task file are used:

```
$ cd cassandra
$ tree
.
├── tasks
│   ├── cassandra.yml
│   ├── jdk.yml
│   └── main.yml
├── templates
│   └── cassandra.yaml
└── vars
    └── main.yml

3 directories, 5 files
```

The `cassandra` role also looks similar to the `build_agent` role in terms of structure. Let's look at the `main.yml` and `jdk.yml` files to start with. Consider the following screenshot:

```
$ cat cassandra/tasks/main.yml
---
- include: jdk.yml
- include: cassandra.yml
$ cat cassandra/tasks/jdk.yml
---
  - name: download jdk rpm
    get_url: url=http://{{ fileserver }}/{{ jdk6 }} dest=/tmp thirsty=no
    tags:
      - jdk

  - name: setup java jdk
    yum: name=/tmp/{{ jdk6 }} state=present
    tags:
      - jdk
```

Let's now look at `cassandra.yml` of the `cassandra` role before proceeding with the explanations.

```
$ cat cassandra/tasks/cassandra.yml
---
  - name: create cassandra directory
    file: dest={{ cassandra_base_directory }} state=directory owner=root group=root
    tags:
      - cassandra_config

  - name: download cassandra
    get_url: url=http://{{ fileserver }}/{{ cassandra_software }} dest={{ cassandra_base_directory }} thirsty=no
    tags:
      - cassandra_config

  - name: untar cassandra
    shell: cd {{ cassandra_base_directory }}; tar -zxvf {{ cassandra_software }} creates={{ cassandra_base_directory }}/{{ cassandra_directory
}}
    tags:
      - cassandra_config

  - name: setup cassandra yml
    template: src=cassandra.yaml dest={{ cassandra_base_directory }}/{{ cassandra_directory }}/conf
    tags:
      - cassandra_config

  - name: symlink cassandra
    file: src={{ cassandra_base_directory }}/{{ cassandra_directory }} dest={{ cassandra_base_directory }}/apache-cassandra owner=root group=ro
ot state=link
    tags:
      - cassandra_config
```

The `main.yml` file of the `cassandra` role includes `jdk.yml` and `cassandra.yml`. The `jdk.yml` file is an exact copy of what we saw earlier; `cassandra.yml` basically has tasks to install Cassandra and set up the required files.

> This might not be the best way to install Cassandra, especially if you expect Cassandra to run as a service, but it solves our purpose. As part of the build step, we've seen folks that have an extra step that explicitly starts up Cassandra (or the database they choose), runs the tests on the `build_agent` machine, and then stops Cassandra (their database). You're free to tweak it as per your environment and convenience.

In the `build_agent` role section, we had a `site.yml` file that we passed as a parameter to the `ansible-playbook` command. Before adding the role to `site.yml`, we can test this role separately by creating a separate `cassandra.yml` file, as shown in the following screenshot:

```
$ cat cassandra.yml
- hosts: cassandra_nodes
  user: vagrant
  sudo: yes
  roles:
    - cassandra
  tags:
    - cassandra
$ cat inventory
[build_agents]
192.168.33.10
[cassandra_nodes]
192.168.33.11
```

The preceding `cassandra.yml` file creates a separate standalone file that contains mapping between hosts and roles. Similarly, if we have a new role that can run on its own, we can create a new YAML file, tying the hosts and the role in this YAML file. You must be wondering what happened to `site.yml`. We'll come to this, but before that, with our new found `cassandra.yml`, let's run the `ansible-playbook` command as follows:

```
$ ansible-playbook -i inventory --private-key ~/.ssh/ansible_key cassandra.yml

PLAY [cassandra_nodes] ********************************************************

GATHERING FACTS **************************************************************
ok: [192.168.33.11]

TASK: [cassandra | download jdk rpm] ****************************************
changed: [192.168.33.11]

TASK: [cassandra | setup java jdk] ******************************************
changed: [192.168.33.11]

TASK: [cassandra | create cassandra directory] ****************************
changed: [192.168.33.11]

TASK: [cassandra | download cassandra] ************************************
changed: [192.168.33.11]

TASK: [cassandra | untar cassandra] ***************************************
changed: [192.168.33.11]

TASK: [cassandra | setup cassandra yml] ***********************************
changed: [192.168.33.11]

TASK: [cassandra | symlink cassandra] *************************************
changed: [192.168.33.11]

PLAY RECAP ****************************************************************
192.168.33.11                : ok=8    changed=7    unreachable=0    failed=0
```

As you would have expected, it ran against the `cassandra_nodes` group of the inventory file that consists of exactly one IP, currently, `192.168.33.11`.

Let's now get back to `site.yml`. This will be the master YAML file that will have details around how Ansible will work against your inventory when you have roles. We can have the `site.yml` file constructed in multiple ways as follows:

- The first way is to have a single monolithic file with multiple plays. In our case, we have two plays.

```
$ cat site.yml
- hosts: build_agents
  user: vagrant
  sudo: yes
  roles:
   - build_agent
  tags:
   - build_agent
- hosts: cassandra_nodes
  user: vagrant
  sudo: yes
  roles:
   - cassandra
  tags:
   - cassandra
```

- After hearing terms such as monolithic, your mind must have alerted you regarding the approach. Hence, we now have a better approach where you use the `site.yml` file to include multiple files, which in turn represent a single `play`. This also allows you to run specific commands against each file (like we saw with the `cassandra.yml` example). This approach also allows a better and faster way of debugging in case a certain play has an issue. The demonstration of this approach is shown in the following screenshot:

```
$ cat site.yml
---
- include: build_agent.yml
- include: cassandra.yml
$
$ cat build_agent.yml
---
- hosts: build_agents
  user: vagrant
  sudo: yes
  roles:
   - build_agent
  tags:
   - build_agent
$
$ cat cassandra.yml
- hosts: cassandra_nodes
  user: vagrant
  sudo: yes
  roles:
   - cassandra
  tags:
   - cassandra
```

If you want to configure your entire infrastructure and you're using roles, you should be in a position to do so by running ansible-playbook with site.yml as a parameter. If you're able to achieve that, then you've done a good job with your infrastructure. Ansible will follow the order that is specified in the site.yml file. So in this case, it will first configure the build_agent node and then the cassandra node. If you have 10 nodes under build_agents, it will configure all of them before moving to the cassandra nodes. Let's see how the Ansible run spans out with the site.yml file in our current case.

```
$ ansible-playbook -i inventory --private-key ~/.ssh/ansible_key site.yml

PLAY [build_agents] ************************************************************

GATHERING FACTS ***************************************************************
ok: [192.168.33.10]

TASK: [build_agent | download jdk rpm] ****************************************
ok: [192.168.33.10]

TASK: [build_agent | setup java jdk] ******************************************
ok: [192.168.33.10]

TASK: [build_agent | create ant directory] ************************************
ok: [192.168.33.10]

TASK: [build_agent | download ant] ********************************************
ok: [192.168.33.10]

TASK: [build_agent | untar ant] ***********************************************
skipping: [192.168.33.10]

TASK: [build_agent | add ant to /etc/profile.d] *******************************
ok: [192.168.33.10]

TASK: [build_agent | download maven] ******************************************
ok: [192.168.33.10]

TASK: [build_agent | untar maven] *********************************************
skipping: [192.168.33.10]

TASK: [build_agent | add maven file to /etc/profile.d] ************************
ok: [192.168.33.10]

PLAY [cassandra_nodes] ********************************************************

GATHERING FACTS ***************************************************************
ok: [192.168.33.11]

TASK: [cassandra | download jdk rpm] ******************************************
ok: [192.168.33.11]

TASK: [cassandra | setup java jdk] ********************************************
ok: [192.168.33.11]

TASK: [cassandra | create cassandra directory] *******************************
ok: [192.168.33.11]

TASK: [cassandra | download cassandra] ****************************************
ok: [192.168.33.11]

TASK: [cassandra | untar cassandra] *******************************************
skipping: [192.168.33.11]

TASK: [cassandra | setup cassandra yml] ***************************************
ok: [192.168.33.11]

TASK: [cassandra | symlink cassandra] *****************************************
ok: [192.168.33.11]
```

The result is as we had explained before. It runs one role at a time and configures all the hosts assigned to that role till it completes the configuration of the entire infrastructure.

If we have a large number of roles, it will become difficult to understand what tasks each role will run. We've covered in *Chapter 1*, *Getting Started with Ansible*, how we can list all the tasks that are going to run and we should use this option now. Though, the question is, do you remember how you can view all the tasks associated with a playbook or role? We're hoping that you will instantly say `list-tasks`.

Let's run it against `cassandra.yml` as follows:

```
$ ansible-playbook -i inventory --private-key ~/.ssh/ansible_key cassandra.yml --list-tasks

playbook: cassandra.yml

  play #1 (cassandra_nodes):
    download jdk rpm
    setup java jdk
    create cassandra directory
    download cassandra
    untar cassandra
    setup cassandra yml
    symlink cassandra
```

The output in the preceding screenshot shows you a single play along with all the associated tasks. Let's now run it against `site.yml` and you should expect two plays as follows:

```
$ ansible-playbook -i inventory --private-key ~/.ssh/ansible_key site.yml --list-tasks

playbook: site.yml

  play #1 (build_agents):
    download jdk rpm
    setup java jdk
    create ant directory
    download ant
    untar ant
    add ant to /etc/profile.d
    download maven
    untar maven
    add maven file to /etc/profile.d

  play #2 (cassandra_nodes):
    download jdk rpm
    setup java jdk
    create cassandra directory
    download cassandra
    untar cassandra
    setup cassandra yml
    symlink cassandra
```

Bingo! The outputs that we view are quite helpful for anyone running the tasks to view what plays will be run and what the associated tasks with each play are.

You must have seen that we've been using tags for all tasks. If we want to run specific tasks, we can use tags. We'll show two examples as follows:

- To run only the `cassandra` setup tasks from `cassandra.yml`, perform the steps shown in the following screenshot:

```
$ ansible-playbook -i inventory --private-key ~/.ssh/ansible_key cassandra.yml --tags=cassandra

PLAY [cassandra_nodes] ******************************************************

GATHERING FACTS *************************************************************
ok: [192.168.33.11]

TASK: [cassandra | create cassandra directory] *****************************
ok: [192.168.33.11]

TASK: [cassandra | download cassandra] *************************************
ok: [192.168.33.11]

TASK: [cassandra | untar cassandra] ***************************************
skipping: [192.168.33.11]

TASK: [cassandra | setup cassandra yml] ***********************************
ok: [192.168.33.11]

TASK: [cassandra | symlink cassandra] *************************************
ok: [192.168.33.11]

PLAY RECAP *****************************************************************
192.168.33.11              : ok=5    changed=0    unreachable=0    failed=0
```

- To run only the `build_agent` role tasks when we are running with the `site.yml` file, perform the steps shown in the following screenshot:

```
$ ansible-playbook -i inventory --private-key ~/.ssh/ansible_key site.yml --tags=build_agent

PLAY [build_agents] ********************************************************

GATHERING FACTS ************************************************************
ok: [192.168.33.10]

TASK: [build_agent | download jdk rpm] ************************************
ok: [192.168.33.10]

TASK: [build_agent | setup java jdk] **************************************
ok: [192.168.33.10]

TASK: [build_agent | create ant directory] *******************************
ok: [192.168.33.10]

TASK: [build_agent | download ant] ***************************************
ok: [192.168.33.10]

TASK: [build_agent | untar ant] ******************************************
skipping: [192.168.33.10]

TASK: [build_agent | add ant to /etc/profile.d] **************************
ok: [192.168.33.10]

TASK: [build_agent | download maven] *************************************
ok: [192.168.33.10]

TASK: [build_agent | untar maven] ****************************************
skipping: [192.168.33.10]

TASK: [build_agent | add maven file to /etc/profile.d] *******************
ok: [192.168.33.10]

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

Let's now go back to our `Vagrantfile`. We now have a modified `Vagrantfile` and we've added a second machine, where each machine will be configured with one role. The `ansible.playbook` parameter now refers to the `build_agent` role for the first machine and the `cassandra` role for the second; the following screenshot demonstrates this:

```
$ cat Vagrantfile
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "devbase"
  config.vm.define :node1 do |machine|
    machine.vm.network :private_network, ip: "192.168.33.10"
    machine.vm.provision "ansible" do |ansible|
      ansible.inventory_path = "inventory"
      ansible.playbook = "build_agent.yml"
      ansible.sudo = true
      ansible.limit = 'all'
    end
  end
  config.vm.define :node2 do |machine|
    machine.vm.network :forwarded_port, guest: 80, host: 8081
    machine.vm.network :private_network, ip: "192.168.33.11"
    machine.vm.provision "ansible" do |ansible|
      ansible.inventory_path = "inventory"
      ansible.playbook = "cassandra.yml"
      ansible.sudo = true
      ansible.limit = 'all'
    end
  end
end
```

We'll now run `vagrant provision` only on the new node, just to see whether provisioning works as expected.

```
$ vagrant provision node2
[node2] Running provisioner: ansible...

PLAY [cassandra_nodes] ********************************************************

GATHERING FACTS **************************************************************
ok: [192.168.33.11]

TASK: [cassandra | download jdk rpm] ****************************************
changed: [192.168.33.11]

TASK: [cassandra | setup java jdk] ******************************************
changed: [192.168.33.11]

TASK: [cassandra | create cassandra directory] *****************************
changed: [192.168.33.11]

TASK: [cassandra | download cassandra] *************************************
changed: [192.168.33.11]

TASK: [cassandra | untar cassandra] ***************************************
changed: [192.168.33.11]

TASK: [cassandra | setup cassandra yml] ***********************************
changed: [192.168.33.11]

TASK: [cassandra | symlink cassandra] *************************************
changed: [192.168.33.11]

PLAY RECAP ****************************************************************
192.168.33.11                 : ok=8    changed=7    unreachable=0    failed=0
```

Yes, it did! If you're wondering about tests, we'll leave that as a task for you
to complete.

Creating a task file with roles

Everything worked as expected. "What's the problem?", you might ask. The problem
with the preceding modeling is that jdk.yml is a duplicate in both roles. If we have
to follow the DRY model, we have to now have a new role to install Java and then
add that role as a dependency to the build_agent and cassandra roles. Let's see
how we can do this.

First, we'll add the `jdk` role. Let's see the configuration in the following screenshot:

```
$ tree jdk
jdk
├── tasks
|   └── main.yml
└── vars
    └── main.yml

2 directories, 2 files
$ cat jdk/tasks/main.yml
---
  - name: download jdk rpm
    get_url: url=http://{{ fileserver }}/{{ jdk6 }} dest=/tmp thirsty=no
    tags:
      - jdk

  - name: setup java jdk
    yum: name=/tmp/{{ jdk6 }} state=present
    tags:
      - jdk
$ cat jdk/vars/main.yml
fileserver: 192.168.1.4
jdk7: jdk-7u51-linux-x64.rpm
jdk6: jdk-6u45-linux-amd64.rpm
```

We now look at a new option that helps us define role dependencies. We define `meta/main.yml` in the `role` folder to define `jdk` as a dependency for the `build_agent` and the `cassandra` roles, shown as follows:

```
$ cat cassandra/meta/main.yml
---
dependencies:
 - { role: jdk }


$ cat build_agent/meta/main.yml
---
dependencies:
 - { role: jdk }
```

You will also see that the `include` tasks do not include `jdk.yml` anymore. The `build_agent` and the `cassandra` roles, are given as follows:

```
$ cat build_agent/tasks/main.yml
---
- include: ant.yml
- include: maven.yml


$ cat cassandra/tasks/main.yml
---
- include: cassandra.yml
```

Let's run it against `build_agent.yml` to see whether the changes get reflected.

```
$ ansible-playbook -i inventory --private-key ~/.ssh/ansible_key build_agent.yml

PLAY [build_agents] *******************************************************

GATHERING FACTS ***********************************************************
ok: [192.168.33.10]

TASK: [jdk | download jdk rpm] ********************************************
changed: [192.168.33.10]

TASK: [jdk | setup java jdk] **********************************************
changed: [192.168.33.10]

TASK: [build_agent | create ant directory] *******************************
ok: [192.168.33.10]

TASK: [build_agent | download ant] ***************************************
changed: [192.168.33.10]

TASK: [build_agent | untar ant] ******************************************
changed: [192.168.33.10]

TASK: [build_agent | add ant to /etc/profile.d] **************************
changed: [192.168.33.10]

TASK: [build_agent | download maven] *************************************
changed: [192.168.33.10]

TASK: [build_agent | untar maven] ****************************************
changed: [192.168.33.10]

TASK: [build_agent | add maven file to /etc/profile.d] *******************
changed: [192.168.33.10]

PLAY RECAP ***************************************************************
192.168.33.10              : ok=10    changed=8    unreachable=0    failed=0
```

In the preceding case, even though we've not used JDK separately, we can do so anytime. Later, if we introduce any other component that is dependent on JDK and maybe a different version of Java 6, we can even override the value at the role dependency level shown as follows:

```
---

dependencies:
  - { role: jdk, jdk6: 'jdk-6u51-linux-amd64.rpm' }
```

All other ways of overriding the variables that we've already discussed also apply.

Using handlers with roles

Let's look at one final example where we'll see how we can use handlers in the role. For this, we add a service script to the `Cassandra` role to start up Cassandra and register a handler for the service restart.

We'll start by looking at the new additions to the `cassandra.yml` task file as follows:

```
- name: setup cassandra yml
  template: src=cassandra.yaml dest={{ cassandra_base_directory }}/{{ cassandra_directory }}/conf
  notify:
    - Restart Cassandra
  tags:
    - cassandra_config

- name: create cassandra group
  group: name=cassandra state=present
  tags:
    - cassandra_config

- name: create cassandra user
  user: name=cassandra comment="cassandra user" group=cassandra
  tags:
    - cassandra_config

- name: copy service init script to /etc/init.d
  template: src=cassandra-init.d dest=/etc/init.d/cassandra owner=root group=root mode=0755
  tags:
    - cassandra_config
  sudo: yes

- name: start cassandra service
  service: name=cassandra state=started
  tags:
    - cassandra_config
```

The handler configuration is given as follows:

```
$ cat cassandra/handlers/main.yml
- name: Restart Cassandra
  action: service name=cassandra state=restarted
```

Like in the other cases, we have a folder named `handlers` and a `main.yml` file, which is the default file where we provide the configuration for the `Restart Cassandra` handler.

The overall tree structure of the `cassandra` directory now looks as follows:

```
$ tree cassandra
cassandra
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── tasks
│   ├── cassandra.yml
│   ├── jdk.yml
│   └── main.yml
├── templates
│   ├── cassandra-init.d
│   └── cassandra.yaml
└── vars
    └── main.yml


5 directories, 8 files
```

We'll leave it to you as an exercise to write tests to check whether the Cassandra service is up and running. This is, in fact, a perfect case to test, simply because the Ansible run might say that the service is up, but it might have resulted in an error that might have prevented the service from starting up. This typically should never happen but we've seen situations where due to `init` script errors, the process might go down a minute after you start it but Ansible or any other tool would report them as `up`.

We have seen how to model roles for your infrastructure and urge you to try it out for your case. Next, we will look at how Jinja2 filters can be used as part of templates.

The Ansible template – Jinja filters

Templates allow you to dynamically construct your playbook and its related data such as variables and other data files. Ansible uses the Jinja2 templating language. We have already covered the template basics in *Chapter 1*, *Getting Started with Ansible*. In this section, we will move forward and see how Jinja2 filters work with Ansible.

Jinja2 filters are simple Python functions that take some arguments, process them, and return the result. For example, consider the following command:

```
{{ myvar | filter }}
```

In the preceding example, `myvar` is a variable; Ansible will pass `myvar` to the Jinja2 filter as an argument. The Jinja2 filter will then process it and return the resulting data. Jinja2 filters even accept additional arguments as follows:

```
{{ myvar | filter(2) }}
```

In this example, Ansible will now pass two arguments, that is, `myvar` and `2`. Likewise, you can pass multiple arguments to filters separated by commas.

Ansible supports a wide variety of Jinja2 filters, we will see some of the important Jinja2 filters that you might need to use while writing your playbook.

Formatting data using filters

Ansible supports Jinja2 filters to format data to JSON or YAML. You pass a dictionary variable to this filter, and it will format your data into JSON or YAML: for example, consider the following command line:

```
{{ users | to_nice_json }}
```

In the preceding example, `users` is the variable and `to_nice_json` is the Jinja2 filter. As we saw earlier, Ansible will internally pass users as an argument to the Jinja2 filter `to_nice_json`. Likewise, you can format your data into YAML as well, using the following command:

```
{{ users | to_nice_yaml }}
```

Using filters with conditionals

You can use Jinja2 filters with conditionals for checking if the status of a task is `failed`, `changed`, `success`, or `skipped`. Let's start looking at examples:

```
- name: Checking httpd service
  service: name=httpd state=running
  register: httpd_result
  ignore_errors: true

- debug: msg="Previous task failed"
  when: httpd_result|failed
```

In the preceding example, we first checked whether the `httpd` service was running and stored the output of that module in the `httpd_result` variable. We then checked whether the previous task failed using the Jinja2 filter, `httpd_result|failed`. Ansible will skip this task if the `when` condition fails, that is, if the previous task passed. Likewise, you can use `changed`, `success`, or `skipped` filters.

Defaulting undefined variables

Instead of failing an Ansible run when a variable is not defined, you can assign a default value to that variable, for example, a value similar to the following:

`{{ backup_disk | default("/dev/sdf") }}`

This filter will not assign the default value to the variable; it will only pass the default value to the current task where it is being used. Let's look at a few more examples of Jinja filters themselves before closing this section:

- Using Random Number filters: To find a random number, character, or string out of a list, you can use the `random` filter:
    - Execute this to get a random character from a list:

      `{{ ['a', 'b', 'c', 'd'] | random }}`

    - Execute this to get a random number from 0 to 100:

      `{{ 100 | random }}`

    - Execute this to get a random number from 10 to 50:

      `{{ 50  | random(10) }}`

    - Execute this to get a random number from 20 to 50 in steps of 10:

      `{{ 50  | random(20, 10) }}`

- Concatenating a list to string using filters: Jinja2 filters allow you to concatenate a list to a string using the `join` filter. This filter takes a separator as an extra argument. If you do not specify any separator, then the filter will combine all elements of the list together without any separation. Consider the following example:

```
{{ ["This", "is", "a", "string"] | join(" ") }}
```

  The preceding filter will result in a `This is a string` output. You can specify any separator you want instead of a white space.

- Encoding or decoding data using filters: You can encode or decode data using filters as follows:
    - Encode your data to base64 using the `b64encode` filter:

      ```
      {{ variable | b64encode }}
      ```

    - Decode an encoded base64 string using the `b64decode` filter:

      ```
      {{ "aGFoYWhhaGE=" | b64decode }}
      ```

Security Management

The last section in this chapter is around Security Management. If you tell your sysadmin that you want to introduce a new feature or a tool, one of the first questions they would ask you would be, what security feature(s) are present with your tool? We'll try to answer these questions from an Ansible perspective in this section. Let's look at them in greater detail.

Using Ansible Vault

Ansible Vault is an exciting feature of Ansible that was introduced in Ansible Version 1.5. This allows you to have encrypted passwords as part of your source code. A recommended practice is to *NOT* have passwords in plain text as part of your repository, especially because anyone who checks out your repository can view your passwords. Let's see how `ansible-vault` can help you with this in the following screenshot:

```
$ ansible-vault create passwords.yml
Vault password:
Confirm Vault password:
$ cat passwords.yml
$ANSIBLE_VAULT;1.1;AES256
66356562313233663061303631313130663463663438653066393764653431366236663834363332
62323361646432303131313564376137373633366663393533640a636466638326462306230139303639
33626539363436363933393326534383331636332623964373166313337316261336236636434343233
63373261393439961610a656264633661393636383238346562388316563323137656261313139626566
31636232663434393534343836323936339633936339346231613065386161626430313265613662666333
653335373430656663133663938343465616265353306330396665
$ ansible-vault edit passwords.yml
Vault password:
$ ansible-vault decrypt passwords.yml
Vault password:
Decryption successful
$ cat passwords.yml
This is a password protected file
```

With reference to the preceding snapshot, for the sake of understanding, we initially created a file using `ansible-vault` named `passwords.yml`. It prompted us for a password. On entering a password, it took us to the editor where we entered `This is a password protected file`. After exiting from the file, we then tried to view the file, only to see that we couldn't view it. Phew! Instead, we saw the file encrypted with AES256, which is the default cipher. We then wanted to edit the file and in that case as well, it prompted us for a password. Finally, we wanted to see the actual content, and we ran the `decrypt` option on the file, and on entering the password, it showed us the content.

Let's now see the next set of options in the following screenshot:

```
$ cat importantfile
this is an important file
$ ansible-vault encrypt importantfile --vault-password-file .password
Encryption successful
$ cat importantfile
$ANSIBLE_VAULT;1.1;AES256
64613564323035643064373436333306538653265333537376433643436393839363030353465353539
32656233383261666438633263339653634643733653362310a64353236633533362316539937313138
39303531333237343538323265626335376165306233393562646162663665326534323332366361
6336666662303561310a61336664656534346163373431376438623037636333362323766356633135
326436613235626262353264623363533336316561396530326633306316539934616466
$ ansible-vault decrypt importantfile --vault-password-file .password
Decryption successful
$ cat importantfile
this is an important file
```

For existing files, we can encrypt them as we've done with the preceding `importantfile` file. However, in the preceding case, we've used the `--vault-password-file` option. This option allows us to have the password that is going to be used for encryption as a single line in a particular file. So, in our case, we have a `.password` file that can be used to encrypt the file. Similarly, you can decrypt the file as well. This is especially useful when you want to decrypt files on the fly from CI or during deployment after checking out code that has encrypted files in it. Users familiar with Chef can view this option as being similar to encrypted data bags.

You can also use Vault with `ansible-playbook`. You'll need to decrypt the file on the fly using a command such as the following:

```
$ ansible-playbook site.yml --vault-password-file .password
```

There is yet another option that allows you to decrypt files using a script, which can then look up some other source and decrypt the file. This can also be a useful option to provide more security. However, make sure that the `get_password.py` script has executable permissions.

```
$ ansible-playbook site.yml --vault-password-file ~/.get_password.py
```

You might be wondering where the `.password` file came from. This file needs to be present on the command center in a location that is accessible to the users who need to decrypt the password file while running the playbook. We typically recommend you to add it to `/opt/.password` and check out your Ansible repository to `/opt/<ansible-repo>`. You can create the `.password` file at the time of the startup. The `.` character in the `.password` file is to make sure that the file is hidden by default when you look for it. This is similar to the MySQL root password that is created in newer versions in `/root/.password` instead of a more visible password file.

The `.password` file content should either be a password or a key that is secure and accessible only to folks who have permission to run commands on the command center. Finally, make sure that you're not encrypting every file that's available! Vault should be used only for important information that needs to be secure.

Encrypting user passwords

Vault takes care of passwords that are checked in and helps you handle them while running Ansible playbooks or commands. However, when Ansible plays are run, at times you might need your users to enter passwords. You also want to make sure that these passwords don't appear in the comprehensive Ansible logs (the default location: `/var/log/ansible.log`) or on `stdout`.

Ansible uses `Passlib`, which is a password-hashing library for Python, to handle encryption for prompted passwords. You can use any of the following algorithms supported by `Passlib`.

- `des_crypt` - DES Crypt
- `bsdi_crypt` - BSDi Crypt
- `bigcrypt` - BigCrypt
- `crypt16` - Crypt16
- `md5_crypt` - MD5 Crypt
- `bcrypt` - BCrypt
- `sha1_crypt` - SHA-1 Crypt
- `sun_md5_crypt` - Sun MD5 Crypt
- `sha256_crypt` - SHA-256 Crypt
- `sha512_crypt` - SHA-512 Crypt
- `apr_md5_crypt` - Apache's MD5-Crypt variant
- `phpass` - PHPass' Portable Hash
- `pbkdf2_digest` - Generic PBKDF2 Hashes
- `cta_pbkdf2_sha1` - Cryptacular's PBKDF2 hash
- `dlitz_pbkdf2_sha1` - Dwayne Litzenberger's PBKDF2 hash
- `scram` - SCRAM Hash
- `bsd_nthash` - FreeBSD's MCF-compatible nthash encoding

Let's now see how encryption works with a variable prompt.

```
vars_prompt:

 - name: "ssh_password"
   prompt: "Enter ssh password"
   private: yes
   encrypt: "md5_crypt"
   confirm: yes
   salt_size: 7
```

In the preceding screenshot, `vars_prompt` is used to prompt users for some data.

The `name` module indicates the actual variable name where Ansible will store the user password, as shown in the following command:

`name: "ssh_password"`

We are using the `prompt` module to prompt users for the password as follows:

prompt: "Enter ssh password"

We are explicitly asking Ansible to hide the password from `stdout` by using `private`; this works like any other password prompt in UNIX systems. The `private` module is accessed as follows:

private: yes

We are using the `md5_crypt` algorithm over here with a salt size of `7`:

encrypt: "md5_crypt"

salt_size: 7

Moreover, Ansible will prompt for the password twice and compare both passwords:

confirm: yes

Hiding passwords

Ansible, by default, filters output that contains the `login_password` key, the `password` key, and the `user:pass` format. For example, if you are passing a password in your module using `login_password` or the `password` key, then Ansible will replace your password with `VALUE_HIDDEN`. Let's now see how you can hide a password using the `password` key:

```
- name: Running myscript.sh
  shell: myscript.sh password=mypass
```

In the preceding shell script, we use the `password` key to pass passwords. This will allow Ansible to hide it from `stdout` and its log file.

Now, when you run the preceding task in the verbose mode, you should not see your `mypass` password; instead Ansible, with `VALUE_HIDDEN`, will replace it as follows:

<web001> REMOTE_MODULE command myscript.sh password=VALUE_HIDDEN #USE_ SHELL

Using no_log

Ansible will hide your passwords only if you are using a specific set of keys. However, this might not be the case every time; moreover, you might also want to hide some other confidential data. The `no_log` feature of Ansible will hide your entire task from logging it to the `syslog` file. It will still print your task on `stdout` and log it to other Ansible logfiles.

> At the time of writing this book, Ansible did not support hiding tasks from `stdout` using `no_log`.

Let's now see how you can hide an entire task with `no_log` as follows:

```
- name: Running myscript.sh
  shell: myscript.sh password=mypass
  no_log: true
```

By passing `no_log: true` to your task, Ansible will prevent the entire task from hitting syslog.

Summary

With this, we come to the end of this chapter, where we've seen how to use features such as `local_action`, conditionals, loops, and including of files to reduce redundant code. We also saw how to use handlers, how to model infrastructure using roles, how to use Jinja filters, and finally security management with Ansible. We now have made significant progress in terms of our Ansible skillset.

We'll next look at how to handle errors, send notifications, even run rollbacks, and utilize callbacks in our Ansible code. This will help us think of our operations a lot more and make sure that they understand what is going on. As you've just read through what we would term as a "heavy" chapter, you deserve a quick coffee break before proceeding to the next chapter. We also want you to think about the following during your break:

- How will you go about identifying roles for your infrastructure?
- How will you manage SSH keys for users across systems using Ansible?
- What are the possible scenarios in your environment where you would use the `local_action` feature?

4

Error Handling, Rollback, and Reporting

"If you shut the door to all errors, truth will be shut out."

- Rabindranath Tagore

The quote here applies to everything in life and not just infrastructure! However, since we're going to continue talking about infrastructure with a key focus on error handling, rollback, and alerting in this chapter, we thought we'd start the chapter on a philosophical note.

So far, we've seen how to write production-ready Ansible playbooks from a configuration management perspective and test the same to overcome any failure. Any time you write code, it's important to think of error handling, logging, alerting, and reporting and if it's based on infrastructure, it becomes even more important. Error handling helps you control the flow of your automation and allows you to notify your users if something goes wrong. Rollbacks become important when the intended task fails and you want to restore the state of the automation to a well-known stable state. Reporting and alerting go hand in hand and we'll talk about it more when we reach that section. We'll also introduce a powerful concept called **Callbacks** and see how helpful they can be when we run our infrastructure with Ansible.

In this chapter, we will cover the following topics:

- Error handling and Rollback
- Callback plugins: a reporting example
- Monitoring and alerting

Error handling and Rollback

Automation always comes with a wide range of advantages such as increased productivity, better control of systems, and solid workflows. However, with advantages, there are always disadvantages! You need to have good control over errors that might occur during automation. While you're writing the functional code, to overcome such situations you would generally use a try-catch technique, where you try to run your code, catch errors when your code fails, try to handle them gracefully, and if possible, continue with the flow. Ansible doesn't provide a try-catch feature; rather, it leaves it up to the user to handle errors. For certain types of tasks, you can write custom modules that can leverage the power of your programming language's error handling (We will look at custom modules in the next chapter). However, for other Ansible core modules, you can use Ansible's conditional statements or ignore errors (as was shown in the earlier chapters).

Suppose you have a playbook with many tasks. There is a possibility that a certain task might fail. If this is one of the initial tasks where you validate certain prerequisites, then error handling might not make sense. For example, let's say you're dealing with one of the cloud systems (for example, **Amazon Web Services** (**AWS**)) and you have a policy within your organization that whoever utilizes any Ansible playbook to perform cloud-related activities should expose their access keys via environment variables. Hence, as part of your Ansible role or playbook, you might have a `common` or `prerequisite` playbook or role that will check whether your access keys are available as environment variables.

Another prerequisite check that we've often seen is users checking if they have the required `sudo` credentials on machines where the playbooks will run (We've even seen an example related to the `sudo` check in *Chapter 3*, *Taking Ansible to Production*.). This might be required for configuration management tasks and your prerequisite playbook or role would check whether `sudo` access is first enabled. If your tasks fail at this level, then error handling does not make sense!

However, if your playbook fails after certain tasks have been executed, and they've changed the state of the remote system (remember `ok` and `changed`?), then it might cause an outage or unpredictable behavior, as the remote system is not in the intended state. This is definitely a cause for worry if you're one of the key people managing the infrastructure. You need not worry if all tasks were in the `ok` state, as it only means that they were idempotent, but if a task is running for the first time, yes, you need to worry! A classic example might be when certain services don't come up as expected and the machine is already in the load balancer and you end up dropping live traffic.

To handle such errors, we recommend writing a rollback playbook. A rollback playbook will include all the tasks that need to be run if the original playbook fails and restore the playbook to the state from where you can serve traffic. The idea here is to run the primary playbook, and if it fails, run the rollback playbook, which will restore the state on the remote host, resulting in a working system at the end.

Let's consider two practical examples that we've seen from close quarters. Suppose you want to update the `memcached` package on your system, which is being monitored by `monit` (we've already discussed `monit` in *Chapter 3*, *Taking Ansible to Production*). Now, in order to update `memcached`, firstly, you will need to stop the `monit` service so that it doesn't try to start the `memcached` service, and then you will have to update your `memcached` package.

Let's say that when you ran your primary playbook, it successfully stopped `monit` and `memcached`, but failed while it tried to update the `memcached` package. Since the `memcached` and `monit` services are stopped, it will affect what is cached and might cause monitoring alerts due to increased response times on the server. The `memcached` package is the frontend. How do you fix this? Add a rollback playbook. The rollback playbook consists of two tasks: start `memcached` after ensuring the older package is still present on the system, and then start the `monit` service.

Another such example would be in a hosted scenario. Let's say, you have a customer x, who is configured to run behind a load balancer with two equally balanced servers. Now, if one of your engineers initiates a restart on the first server, then the CPU load will spike on the other server. To take care of the spike, let's say another engineer initiates a restart on the second server while the first one is still not serving the traffic. Now, both your servers are down and your entire traffic towards that load balancer will be affected, resulting in a downtime for your customer.

In order to deal with such human errors, you can use locking. Whenever a restart is initiated on one of the servers, make sure that you lock the load balancer, which is the shared resource, using a tool such as `etcd`, and unlock it only after the restart is completed and the server can serve the traffic. In the meantime, no one can restart the second server until the first server releases its lock. There is also an aspect of making sure the traffic isn't unmanageable on the second server in such situations when there are other complexities around it. However, basically, you want to make sure that a shared resource is locked while performing such operations.

> Note that `etcd` is a highly available key-value store similar to Apache ZooKeeper.

Let's see how we can use locking with the Apache Solr restart for multiple servers that are part of the same load balancer, as shown in the following screenshot:

```
$ cat playbooks/solr_restart.yml
---
- hosts: web001
  remote_user: ec2-user
  gather_facts: no

  tasks:

    - name: Trying to obtain lock with the following lockname and lockvalue
      debug: msg="lock name=lb/{{ load_balancer }} lock value={{ inventory_hostname }}"

    - name: Locking {{ load_balancer }}
      local_action: uri url=http://localhost:4001/v2/keys/lb/{{ load_balancer }}?prevExist=false method=PUT body="value={{ inventory_hostname }}" status_code=201 HEADER_Content-Type="application/x-www-form-urlencoded" return_content=yes

    - name: Restarting solr
      service: name=solr state=restarted
      sudo: yes

    - name: Unlocking {{ load_balancer }}
      local_action: uri url=http://localhost:4001/v2/keys/lb/{{ load_balancer }} method=DELETE
```

In the preceding playbook, we first print the lock name and the lock value for readability. We then go ahead and try to lock the load balancer so that no one else can restart any other server behind that load balancer. If the locking task fails, then it means the load balancer is already locked and someone else has restarted one of the Apache Solr servers behind it. However, if we are successful in obtaining the lock, we restart the `solr` service and then unlock the load balancer. Apart from these steps, you can also add some checks to this playbook before unlocking the load balancer to make sure that the service is actually up and ready to serve the production traffic.

Executing the playbook

Consider the following screenshot and let's see how the `solr_restart.yml` playbook works:

```
$ ansible-playbook -i hosts playbooks/solr_restart.yml -e "load_balancer=lb001"

PLAY [web001] *********************************************************

TASK: [Trying to obtain lock with the following lockname and lockvalue] *******
ok: [web001] => {
    "msg": "lock name=lb/lb001 lock value=web001"
}

TASK: [Locking lb001] *********************************************************
ok: [web001]

TASK: [Restarting solr] *********************************************************
changed: [web001]

TASK: [Unlocking lb001] *********************************************************
ok: [web001]

PLAY RECAP *********************************************************
web001                     : ok=4    changed=1    unreachable=0    failed=0
```

As expected, the preceding playbook:

1. Locked the load balancer.
2. Restarted the `solr` service.
3. Unlocked the load balancer.

This was an example of a successful Ansible run. However, what if Ansible was able to acquire the lock but could not start the service, and the playbook run fails as a result? Let's see what happens when the `solr` restart fails. This is demonstrated in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/solr_restart.yml -e "load_balancer=lb001"

PLAY [web001] *********************************************************

TASK: [Trying to obtain lock with the following lockname and lockvalue] *******
ok: [web001] => {
    "msg": "lock name=lb/lb001 lock value=web001"
}

TASK: [Locking lb001] *********************************************************
ok: [web001]

TASK: [Restarting solr] *********************************************************
failed: [web001] => {"failed": true}
msg: Stopping Solr ...
Stopped
Starting Solr ...


FATAL: all hosts have already failed -- aborting

PLAY RECAP *********************************************************
           to retry, use: --limit @/home/rraithatha/solr_restart.retry

web001                     : ok=2    changed=0    unreachable=0    failed=1
```

As you can see, Ansible acquired the lock but could not restart the service. Now, the `solr` service is not running and the load balancer is locked as well, which will not allow anyone else to restart the other servers from the same load balancer. This is where rollback comes into the picture. In order to roll back, you first need to try to restart the service, and then unlock the load balancer. Let's see an example playbook for rollback in the following screenshot:

```
$ cat playbooks/solr_restart_rollback.yml
---
- hosts: web001
  remote_user: ec2-user
  gather_facts: no

  tasks:

    - name: Checking if {{ load_balancer }} is already locked
      local_action: uri url=http://localhost:4001/v2/keys/lb/{{ load_balancer }}
      register: elb_lock
      ignore_errors: true

    - name: Restarting solr
      service: name=solr state=started
      sudo: yes
      ignore_errors: true
      when: elb_lock.status == 200

    - name: Unlocking {{ load_balancer }}
      local_action: uri url=http://localhost:4001/v2/keys/lb/{{ load_balancer }} method=DELETE
      when: elb_lock.value == "{{ inventory_hostname }}"
      when: elb_lock.status == 200
      ignore_errors: true
```

In the preceding playbook, we first check if the lock still exists. We have nothing to do if there is no lock, in which case we will skip all other tasks because it means that someone from the team has already taken care of the previous Ansible failure. However, if the lock still exists, we then try to restart the `solr` service but not allow Ansible to fail, so that we can remove the lock even if the restart fails. You will not allow Ansible to fail if there are multiple servers.

If there are only two servers behind the load balancer and the `solr` service restart fails, you should *NOT* remove the lock, as restarting the second server in this case can result in an outage. You can also notify the user by sending a mail asking them to troubleshoot the issue manually. We will see how mail notification works later in this chapter. Consider the following screenshot where we're running the rollback playbook:

```
$ ansible-playbook playbooks/solr_restart_rollback.yml -i hosts -e "load_balancer=lb001"

PLAY [web001] ***************************************************************

TASK: [Checking if lb001 is already locked] *******************************
ok: [web001]

TASK: [Restarting solr] ***************************************************
changed: [web001]

TASK: [Unlocking lb001] ***************************************************
ok: [web001]

PLAY RECAP ****************************************************************
web001                     : ok=3    changed=1    unreachable=0    failed=0
```

As explained before, Ansible checks if the lock exists, starts the service, and then removes the lock.

Instead of running the rollback playbook manually on every failure, you can write a small wrapper script, which can run the Ansible playbook, and depending on the exit code of the Ansible playbook, it can run the rollback playbook if required. It could be as simple as a bash script, which directly runs the `ansible-playbook` command.

Callback plugins

We now proceed to callback plugins. One of the features that Ansible provides is a **callback mechanism**. You can configure as many callback plugins as required. These plugins can intercept events and trigger certain actions. The section on alerting that we will cover after this section shows you how to use these plugins to make sure you get the right sort of feedback. Let's see a simple example where we just print the run results at the end of the playbook run as part of a callback and then take a brief look at how to configure a callback.

We will rerun the `build_agent` role from the last chapter for this example; the build agent is already configured and we're running Ansible against it. This is shown in the following screenshot:

```
$ ansible-playbook -i inventory --private-key ~/.ssh/ansible_key build_agent.yml

PLAY [build_agents] ********************************************************

GATHERING FACTS ***********************************************************
ok: [192.168.33.10]

TASK: [build_agent | download jdk rpm] ************************************
ok: [192.168.33.10]

TASK: [build_agent | setup java jdk] **************************************
ok: [192.168.33.10]

TASK: [build_agent | create ant directory] *******************************
ok: [192.168.33.10]

TASK: [build_agent | download ant] ***************************************
ok: [192.168.33.10]

TASK: [build_agent | untar ant] ******************************************
skipping: [192.168.33.10]

TASK: [build_agent | add ant to /etc/profile.d] **************************
ok: [192.168.33.10]

TASK: [build_agent | download maven] *************************************
ok: [192.168.33.10]

TASK: [build_agent | untar maven] ****************************************
skipping: [192.168.33.10]

TASK: [build_agent | add maven file to /etc/profile.d] *******************
ok: [192.168.33.10]

PLAY RECAP ***************************************************************
Run Results: {'192.168.33.10': {'unreachable': 0, 'skipped': 2, 'ok': 8, 'changed': 0, 'failures': 0}}
192.168.33.10              : ok=8    changed=0    unreachable=0    failed=0
```

As you can see in the preceding screenshot, the callback plugin resulted in an extra line called `Run Results`, and printed a dictionary or hash of the actual results. You can utilize this information in a number of ways. Isn't this powerful? Are you getting any ideas around how you can utilize a feature such as this? Do write it down before reading further. If you're able to write out even two use cases that we've not covered here and is relevant to your infrastructure, give yourself a pat on the back!

The callback plugin's location is present in `ansible.cfg`. You can use `grep` for the term "callback" in `ansible.cfg`, shown as follows:

```
$ grep callback ansible.cfg
callback_plugins    = /usr/share/ansible_plugins/callback_plugins
```

> The `callback_plugins` path might be different for you depending on how you installed Ansible.

You need to then add your callback scripts in this folder. If you have five callback scripts to perform different operations, you need to add all five of them in the `callback_plugins` folder. The callbacks are in Python and you'll have to know a little bit about Python to make good use of them. You can pretty much copy and paste the following example and get started with the callback plugins.

We have a simple `callback_sample.py` file to start with, the output of which generates the `Run Results` line in the earlier example. Let's look at the folder that's configured in `ansible.cfg`, which is `/usr/share/ansible_plugins/callback_plugins` in our case:

```
$ ls -l /usr/share/ansible_plugins/callback_plugins
callback_sample.py
```

Let's now look at the contents of the `callback_sample` file:

```
$ cat /usr/share/ansible_plugins/callback_plugins/callback_sample.py
class CallbackModule(object):

  def on_any(self, *args, **kwargs):
    pass

  def runner_on_failed(self, host, res, ignore_errors=False):
    pass

  def runner_on_ok(self, host, res):
    pass
```

```python
  def runner_on_error(self, host, msg):
    pass

  def runner_on_skipped(self, host, item=None):
    pass

  def runner_on_unreachable(self, host, res):
    pass

  def runner_on_no_hosts(self):
    pass

  def runner_on_async_poll(self, host, res, jid, clock):
    pass

  def runner_on_async_ok(self, host, res, jid):
    pass

  def runner_on_async_failed(self, host, res, jid):
    pass

  def playbook_on_start(self):
    pass

  def playbook_on_notify(self, host, handler):
    pass

  def playbook_on_no_hosts_matched(self):
    pass

  def playbook_on_no_hosts_remaining(self):
    pass

  def playbook_on_task_start(self, name, is_conditional):
    pass

  def playbook_on_vars_prompt(self, varname, private=True, prompt=None,
encrypt=None, confirm=False, salt_size=None, salt=None, default=None):
    pass

  def playbook_on_setup(self):
    pass

  def playbook_on_import_for_host(self, host, imported_file):
```

```
    pass

  def playbook_on_not_import_for_host(self, host, missing_file):
    pass

  def playbook_on_play_start(self, pattern):
    pass

  def playbook_on_stats(self, stats):
    results = dict([(h, stats.summarize(h)) for h in stats.processed])
    print "Run Results: %s" % results
```

As you can see, the callback class, `CallbackModule`, contains several methods. The methods of this class are called and the Ansible run parameters are provided as parameters to these methods. Playbook activities can be intercepted by using these methods and relevant actions can be taken based on that. Relevant methods are called based on the action, for example, we've used the `playbook_on_stats` method (in bold) to display statistics regarding the playbook run. Let's run a basic playbook with the callback plugin and view the output as follows:

```
$ cat play.yml
---
- hosts: default
  remote_user: vagrant
  sudo: yes
  tasks:
    - name: Test message
      debug: msg="hello"

$ ansible-playbook -i inventory --private-key ~/.ssh/key play.yml

PLAY [default] ********************************************************

GATHERING FACTS *****************************************************
ok: [192.168.33.10]

TASK: [Test message] ************************************************
ok: [192.168.33.10] => {
    "msg": "hello"
}

PLAY RECAP **********************************************************
Run Results: {'192.168.33.10': {'unreachable': 0, 'skipped': 0, 'ok': 2, 'changed': 0, 'failures': 0}}
192.168.33.10              : ok=2    changed=0    unreachable=0    failed=0
```

You can now see the `Run Results` line right at the end, which is due to our custom code. This is just an example of how you can intercept methods and use them to your advantage.

However, there are so many other methods that you can utilize. Spend some time looking at the names of the methods. With all that you learned so far, you must be in a position to recognize or guess what each preceding method might do. The word `pass` indicates no action and is the default value for all these methods that you see. If you find it difficult to understand these methods, you can print the output at the method level and that will provide more food for thought on how you can utilize these methods to your advantage. For example, let's modify the `runner_on_ok` task and print the output that we get from `host` and `res`:

```
def runner_on_ok(self, host, res):

    print 'host - %s , res - %s" % (host,res)
```

On rerunning the playbook, it prints all the facts. A snippet of what we get with the facts is shown below:

```
GATHERING FACTS ***********************************************************
ok: [192.168.33.10]
host - 192.168.33.10 , res - {'invocation': {'module_name': 'setup', 'module_args': ''}, u'verbose_override': True, u'changed': False, u'ansibl
e_facts': {u'facter_operatingsystem': u'CentOS', u'facter_hostname': u'node', u'facter_uptime_hours': 1, u'ansible_distribution_version': u'6.4
', u'facter_sshfp_rsa': u'SSHFP 1 1 bb21e4f734984afade7fec2766cb8250ed3d8753\nSSHFP 1 2 ea255e85e928a3855a9a0eec58234ef88f467367a0feae740fbb7e9
47518b4c8', u'ansible_env': {u'USERNAME': u'root', u'LANG': u'C', u'TERM': u'xterm-256color', u'SHELL': u'/bin/bash', u'SUDO_COMMAND': u'/bin/s
h -c echo SUDO-SUCCESS-prjxwgtgkyagzpvebveulrldbbhzjzdo; /usr/bin/python /home/vagrant/.ansible/tmp/ansible-tmp-1407504738.7-87475440724338/set
up; rm -rf /home/vagrant/.ansible/tmp/ansible-tmp-1407504738.7-87475440724338/ >/dev/null 2>&1', u'SHLVL': u'1', u'SUDO_UID': u'501', u'LC_CTYP
E': u'UTF-8', u'PWD': u'/home/vagrant', u'LOGNAME': u'root', u'USER': u'root', u'MAIL': u'/var/mail/vagrant', u'PATH': u'/sbin:/bin:/usr/sbin:/
usr/bin', u'SUDO_USER': u'vagrant', u'HOME': u'/root', u'SUDO_GID': u'501', u'_': u'/usr/bin/python'}, u'facter_boardproductname': u'VirtualBox
', u'ansible_userspace_bits': u'64', u'ansible_architecture': u'x86_64', u'ansible_default_ipv4': {u'macaddress': u'08:00:27:c9:39:9e', u'netwo
rk': u'10.0.2.0', u'mtu': 1500, u'alias': u'eth0', u'netmask': u'255.255.255.0', u'address': u'10.0.2.15', u'interface': u'eth0', u'type': u'et
her', u'gateway': u'10.0.2.2'}, u'ansible_swapfree_mb': 2559, u'ansible_default_ipv6': {}, u'facter_netmask_eth1': u'255.255.255.0', u'facter_u
uid': u'B4FC6EFC-B7FF-49F9-B5CD-6A820CB00920', u'ansible_cmdline': {u'LANG': u'en_US.UTF-8', u'rd_NO_LUKS': True, u'ro': True, u'KEYBOARDTYPE':
u'pc', u'rd_NO_MD': True, u'quiet': True, u'rd_LVM_LV': u'VolGroup/lv_root', u'rhgb': True, u'KEYTABLE': u'us', u'SYSFONT': u'latarcyrheb-sun1
6', u'root': u'/dev/mapper/VolGroup-lv_root', u'rd_NO_DM': True}, u'ansible_selinux': False, u'facter_bios_vendor': u'innotek GmbH', u'ansible_
userspace_architecture': u'x86_64', u'ansible_product_uuid': u'B4FC6EFC-B7FF-49F9-B5CD-6A820CB00920', u'facter_osfamily': u'RedHat', u'facter_m
```

We see all the facts associated with the remote system given to us on a platter in the form of JSON. This basically means you can even update your inventory system by just running the Ansible `setup` task against all your nodes, parse out the `res` hash or dictionary, and then make the right calls to check or update your inventory system. (By default, if the playbook doesn't have any task, it will just run the `setup` task to get information regarding the nodes. We urge you to try out a playbook with zero tasks but with `gather_facts` not set to `False`. See the result for yourself!) Does this make sense? We strongly urge you to read through this section a couple of times to get a better understanding.

For the Java tasks that we had in the playbook in our previous chapter, the output appears as shown in the following screenshot:

```
TASK: [build_agent | download jdk rpm] ******************************************
ok: [192.168.33.10]
host - 192.168.33.10 , res - {u'src': u'/tmp/tmpnSkXV6', u'md5sum': u'518e6673f3f07e87bbef3e83f287b5f8', u'group': u'root', u'uid': 0, 'item':
'', u'url': u'http://192.168.1.4/jdk-6u45-linux-amd64.rpm', u'changed': False, u'sha256sum': u'', u'dest': u'/tmp/jdk-6u45-linux-amd64.rpm', u'
state': u'file', u'gid': 0, u'mode': u'0644', 'invocation': {'module_name': 'get_url', 'module_args': u'url=http://192.168.1.4/jdk-6u45-linux-a
md64.rpm dest=/tmp thirsty=no'}, u'owner': u'root', u'size': 58763686, u'msg': u'OK (58763686 bytes)'}

TASK: [build_agent | setup java jdk] ******************************************
ok: [192.168.33.10]
host - 192.168.33.10 , res - {u'rc': 0, u'changed': False, u'results': [], 'item': '', 'invocation': {'module_name': 'yum', 'module_args': u'na
me=/tmp/jdk-6u45-linux-amd64.rpm state=present'}, u'msg': u''}
```

Looking at the preceding output, we can conclude that every module that you write in your playbook is queryable or addressable as part of the callback and you can utilize it for better reporting.

Let's see an example where we use callbacks to push information regarding the Ansible playbook run to a MySQL instance for daily, weekly, and monthly reporting, and to analyze what tasks fail the most. You will have to set up your MySQL database and make sure there is connectivity from the command center where the callbacks are being configured. It's time to look at the plugin code, which is as follows:

```
$ cat callback_mysql.py ( remember this callback function should be in
the 'callback_plugins' folder as indicated in ansible.cfg file)
import getpass
import MySQLdb as mdb
import os

class CallbackModule(object):

  def __init__(self):
    self.dbhost = os.getenv('DBHOST')
    self.dbuser = os.getenv('DBUSER')
    self.dbpassword = os.getenv('DBPASSWORD')
    self.dbname = os.getenv('DBNAME')
    self.action = ""
    self.user = ""

  def update_stats(self, host, result, task=None, message=None):
    con = mdb.connect(self.dbhost, self.dbuser, self.dbpassword, self.
dbname)
    cur=con.cursor()
    cur.execute('insert into ansible_run_stats(user,host,res
ult,task,message) values("%s","%s","%s","%s","%s")' %(self.
user,host,result,task,message))
```

```
      con.commit()
      con.close()

  def on_any(self, *args, **kwargs):
    pass

  def runner_on_failed(self, host, res, ignore_errors=False):
      self.update_stats(self.task.play.hosts, "unsuccessful", task=self.
task.name, message=res)

  def runner_on_ok(self, host, res):
    pass

  def runner_on_error(self, host, msg):
    pass

  def runner_on_skipped(self, host, item=None):
    pass

  def runner_on_unreachable(self, host, res):
      self.update_stats(self.task.play.hosts, "unsuccessful", task=self.
task.name, message=res)

  def runner_on_no_hosts(self):
    pass

  def runner_on_async_poll(self, host, res, jid, clock):
    pass

  def runner_on_async_ok(self, host, res, jid):
    pass

  def runner_on_async_failed(self, host, res, jid):
    pass

  def playbook_on_start(self):
    pass

  def playbook_on_notify(self, host, handler):
    pass
```

```
  def playbook_on_no_hosts_matched(self):
    pass

  def playbook_on_no_hosts_remaining(self):
    pass

  def playbook_on_task_start(self, name, is_conditional):
    pass

  def playbook_on_vars_prompt(self, varname, private=True, prompt=None,
encrypt=None, confirm=False, salt_size=None, salt=None, default=None):
    pass

  def playbook_on_setup(self):
    pass

  def playbook_on_import_for_host(self, host, imported_file):
    pass

  def playbook_on_not_import_for_host(self, host, missing_file):
    pass

  def playbook_on_play_start(self, pattern):
    if not self.user:
      self.user = getpass.getuser()

  def playbook_on_stats(self, stats):
    if not stats.dark and not stats.failures:
      self.update_stats(stats.ok.keys()[0], "successful")
```

Let's look at the code in more detail to understand the flow; we first initialize the database parameters by getting them from the environment in the following snippet. We expect the following environment variables to be set: DBHOST, DBUSER, DBPASSWORD, and DBNAME:

```
def  __init__(self):
    self.dbhost = os.getenv('DBHOST')
    self.dbuser = os.getenv('DBUSER')
    self.dbpassword = os.getenv('DBPASSWORD')
    self.dbname = os.getenv('DBNAME')
    self.action = ""
    self.user = ""
```

Since we might have different users who might log in with their account onto the system, we need to make sure we trace who runs what commands. We store the user value in the following method:

```
def playbook_on_play_start(self, pattern):

    if not self.user:

        self.user = getpass.getuser()
```

We then track the status of the playbook run by calling the `update_stats` method (this can either be a class instance method or a method that's outside the class) from the `runner_on_failed`, `runner_on_unreachable`, and `playbook_on_stats` methods to capture the status of the playbook run as follows:

```
def runner_on_failed(self, host, res, ignore_errors=False):

    self.update_stats(self.task.play.hosts, "unsuccessful", task=self.
task.name, message=res)


def runner_on_unreachable(self, host, res):

    self.update_stats(self.task.play.hosts, "unsuccessful", task=self.
task.name, message=res)


def playbook_on_stats(self, stats):

    if not stats.dark and not stats.failures:

        self.update_stats(stats.ok.keys()[0], "successful")
```

Finally, the `update_stats` method connects to the database and pushes the data into a table called `ansible_run_stats`, shown as follows:

```
def update_stats(self, host, result, task=None, message=None):

    con = mdb.connect(self.dbhost, self.dbuser, self.dbpassword, self.
dbname)

    cur=con.cursor()

    cur.execute('insert into ansible_run_stats(user,host,res
ult,task,message) values("%s","%s","%s","%s","%s")' %(self.
user,host,result,task,message))

    con.commit()

    con.close()
```

Let's look at the rows in the database after a few runs. The output is shown in the following screenshot:

```
mysql> select * from ansible_run_stats ;
+----------+--------+--------------+----------------------+-------------------------------------------------------------------------------------------+
| user     | host   | result       | task                 | message                                                                                   |
+----------+--------+--------------+----------------------+-------------------------------------------------------------------------------------------+
| ec2-user | web001 | unsuccessful | Bouncing httpd service | {u'msg': u'Stopping httpd: [FAILED]
Starting httpd: ', u'failed': True, 'invocation': {'module_name': 'service', 'module_args': 'name=httpd state=restarted'}} |
| ec2-user | web001 | successful   | None                 | None                                                                                      |
| ec2-user | web001 | successful   | None                 | None                                                                                      |
| ec2-user | web001 | successful   | None                 | None                                                                                      |
+----------+--------+--------------+----------------------+-------------------------------------------------------------------------------------------+
4 rows in set (0.00 sec)
```

As expected, Ansible recorded all the passed and failed runs' details in the MySQL database. This is a great way to report. Make sure that you dump all the information related to all your runs across all users and run reporting scripts either as cron jobs or from other BI systems to send out reports with information on the number of users, hosts, and tasks that run across your infrastructure using Ansible.

Let's look at the other possible use cases as follows:

- Let's say you're starting a deployment using Ansible and you want to notify a bunch of people about it. In that case, you can utilize the `playbook_on_start` method to alert all of them.
- You can change the way the log output looks for all the tasks.
- You can calculate the overall runtime taken by storing the start time and end time as instance variables.
- You can alert anyone based on each of the outputs. We'll see a couple of examples in the next section

Monitoring and alerting

Monitoring and alerting go hand in hand. At an infrastructure and application level, you end up monitoring several metrics, right from CPU and memory to application-level information such as heap and number of database connections from your application, and alert relevant teams based on that. Also, from an infrastructure automation point of view, you need to make sure you start a strong feedback loop by integrating automation with your monitoring and alerting system or systems through constant reporting.

We'll focus on how you can keep on the Ansible run for any failures or confirmation of task completions rather than staring at your screen continuously. At some stage, you might run all the tasks from a custom UI. It should be fine if you are running a small playbook with a bunch of tasks. However, if it's a long playbook with many tasks, you might derail the flow and miss some of the errors or an important task because of too much data popping out of your `stdout` output. To deal with such a situation, we use **Alerting**. You can notify the user whenever a task/playbook fails or after a task is completed, allowing them to sip their brew, sit back, and relax. There are different types of alerting systems, such as mail, monitoring systems, graphing systems, pagers, chat rooms, and so on. We will look at some of the basic alerting techniques that can be used with Ansible, which are as follows:

- E-mails
- HipChat
- Nagios
- Graphite

E-mails

The easiest and most common way of alerting is to send e-mails. Ansible allows you to send e-mails from your playbook using a `mail` module. You can use this module in between any of your tasks and notify your user whenever required. Also, in some cases, you cannot automate each and every thing because either you lack the authority or it requires some manual checking and confirmation. If this is the case, you can notify the responsible user that Ansible has done its job and it's time for him/her to perform his/her duty. Let's see how you can use the `mail` module to notify your users with one of the example playbooks of *Chapter 3*, *Taking Ansible to Production*, as follows:

```
$ cat playbooks/mail_alert.yml
---
- hosts: web001
  remote_user: ec2-user
  gather_facts: no
  vars:
    users:
      - name: alice
        database:
          - clientdb
          - employeedb
          - providerdb
      - name: bob
        database:
          - clientdb

  tasks:
    - name: give users access to multiple databases
      mysql_user: name={{ item.0.name }} priv={{ item.1 }}.*:ALL append_privs=yes password=foo login_user=root login_password=root
      register: mysql
      with_subelements:
        - users
        - database


    - name: Sending mail to user
      local_action: mail
                    host='127.0.0.1'
                    subject="[Ansible] All task completed for mail_alert playbook"
                    body="Hello, Ansible has done its job and its time for you to perform your duty"
                    from="ansible@example.com"
                    to="user@example.com"
```

In the preceding playbook, we will first loop over the user and database dictionary giving the MySQL users access to the database. If the tasks succeed, we will send an e-mail to the user saying all tasks were completed successfully. Let's see how this works in the following screenshot:
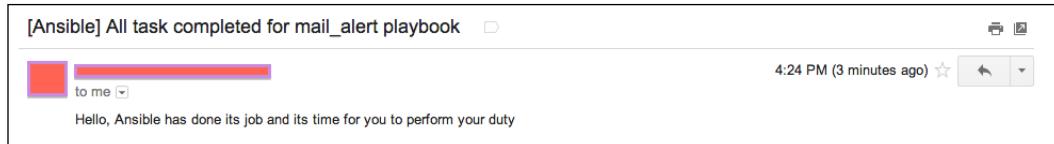
```
$ ansible-playbook -i hosts  mail_alert.yml

PLAY [web001] *****************************************************************

TASK: [give users access to multiple databases] *****************************
ok: [web001] => (item=({'name': 'alice'}, 'clientdb'))
ok: [web001] => (item=({'name': 'alice'}, 'employeedb'))
ok: [web001] => (item=({'name': 'alice'}, 'providerdb'))
ok: [web001] => (item=({'name': 'bob'}, 'clientdb'))

TASK: [Sending mail to user] *************************************************
ok: [web001]

PLAY RECAP *******************************************************************
web001                     : ok=2    changed=0    unreachable=0    failed=0
```

As expected, Ansible tried giving database access to the users, but it found that the users already had access to the database, thus returning with an OK status instead of a CHANGED status. Let's see if they actually received an e-mail. The following screenshot shows an e-mail was sent to the user:

```
[Ansible] All task completed for mail_alert playbook

                                                    4:24 PM (3 minutes ago)
to me

Hello, Ansible has done its job and its time for you to perform your duty
```

Bingo! It worked. Likewise, you can also send the stdout of a task by using variables within the body of your e-mail.

You can also use this module before a prompt task. For example, let's say that you have a playbook that takes around 20–30 minutes to complete, and at the end of the playbook, you ask your user (via a command prompt), whether Ansible should add the host back to the load balancer, since you cannot expect a user to sit in front of a screen for 30 minutes and keep waiting for the prompt to respond. Instead, you can e-mail your users before prompting so that they can get back to the Ansible prompt and do the needful.

HipChat

HipChat is a private group chat and IM network that provides real-time collaboration features for companies and teams. Its usage has grown heavily over the last year or so. Ansible allows you to send notifications to a HipChat room from a playbook using a `hipchat` module. To send a notification, you need to first create a token for the room where you want to notify.

> Refer to the **HipChat** documentation to create API tokens.

Let's see how you can use the `hipchat` module to notify users, as shown in the following screenshot:

```
$ cat hipchat_alert.yml
---
- hosts: web001
  remote_user: ec2-user
  gather_facts: no
  vars:
    users:
      - name: alice
        database:
          - clientdb
          - employeedb
          - providerdb
      - name: bob
        database:
          - clientdb

  tasks:
    - name: give users access to multiple databases
      mysql_user: name={{ item.0.name }} priv={{ item.1 }}.*:ALL append_privs=yes password=foo login_user=root login_password=root
      register: mysql
      with_subelements:
        - users
        - database

    - name: Sending mail to user
      local_action: mail
                    host='127.0.0.1'
                    subject="[Ansible] All task completed for mail_alert playbook"
                    body="Hello, Ansible has done its job and its time for you to perform your duty"
                    from="ansible@yourmmail.com"
                    to="you@yourmmail.com"

    - name: Notifying on hipchat
      hipchat: token=330051dcbc85abeed524d428e7c5d5 room=ops msg="Hello, Ansible has done its job and its time for you to perform your duty" from="Ansible"
```

In the preceding example, we reused the database playbook, which we saw in the *E-mails* section of this chapter. We pass the `hipchat` API token, room name, message, and sender name to the `hipchat` module.

> Note that the `hipchat` module does not support HipChat v2 APIs at the time of writing. You should use a HipChat v1 API with this module.

Let's run the playbook. The output is shown in the following screenshot:

```
$ ansible-playbook -i hosts  hipchat_alert.yml

PLAY [web001] ***********************************************************

TASK: [give users access to multiple databases] ***********************
ok: [web001] => (item=({'name': 'alice'}, 'clientdb'))
ok: [web001] => (item=({'name': 'alice'}, 'employeedb'))
ok: [web001] => (item=({'name': 'alice'}, 'providerdb'))
ok: [web001] => (item=({'name': 'bob'}, 'clientdb'))

TASK: [Sending mail to user] ******************************************
ok: [web001]

TASK: [Notifying on hipchat] ******************************************
changed: [web001]

PLAY RECAP ************************************************************
web001                     : ok=3    changed=1    unreachable=0    failed=0
```
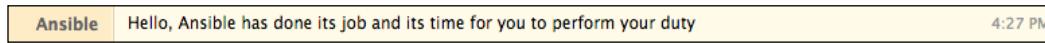
As expected, Ansible gave the database access to the user, sent an e-mail to the user, and notified in the HipChat room. Let's see if we actually received a notification in the `ops` room that we configured. We should receive something like the following screenshot:

| Ansible | Hello, Ansible has done its job and its time for you to perform your duty | 4:27 PM |

Yes we did! We received a notification in our HipChat room. Wasn't that quite easy? Now, it's your job to try out this awesome module.

Nagios

No book or chapter on monitoring or alerting is complete without talking about **Nagios**. Nagios is a widely used, open source monitoring tool, which allows you to monitor almost everything that exists on planet Earth and maybe beyond. Well, jokes apart, you can use Nagios to monitor your playbook run by sending passive checks to the `nagios` server. As we did for the e-mail notifications, you can have a `nagios` task at the end of a playbook, which will notify the `nagios` server if a playbook succeeds; alternatively, you can have a task in the rollback playbook, which will notify the `nagios` server that a playbook has failed.

There are other popular monitoring tools such as Zabbix, Ganglia, and Sensu. You can integrate Ansible in a way similar to what we've shown with Nagios in this example.

Let's see an example playbook that uses a MySQL database and notifies our `nagios` server if everything goes well.

```
$ cat playbooks/nagios_mysql_dump.yml
---
- name: mysql dump
  hosts: web001
  remote_user: ec2-user
  gather_facts: no

  tasks:
    - name: Dumping mysql database
      shell: mysqldump -u {{ lookup('env','DBUSER') }} -p{{ lookup('env','DBPASSWORD') }}
ansible > ansible.sql

    - name: Sending passive check to nagios
      shell: echo -e "web001\tmysql dump\t0\tOK':' Dumped ansible mysql database" | /usr/
sbin/send_nsca -H 10.47.137.69  -c  /etc/nagios/send_nsca.cfg
      sudo: yes
```

In the preceding playbook, we first dump the MySQL table using the `mysqldump` command. If it goes well, we then use a passive check to notify the `nagios` server using the `send_nsca` command. These kinds of backup-and-restore tasks are simple to perform but very often teams forget to do them or do not have the right kind of alerting or monitoring around them and end up wasting a lot of time on them. We felt it's best to show how you can do something as simple as this with Ansible, and at the same time, stress on important best practices such as this in your organization. If you're already doing it, great!

Let's run the playbook. We expect to see an `ok` output in Nagios at the end of the playbook run, as shown in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/nagios_mysql_dump.yml

PLAY [web001] ************************************************************

TASK: [Dumping mysql database] ******************************************
changed: [web001]

TASK: [Sending passive check to nagios] *********************************
changed: [web001]

PLAY RECAP **************************************************************
web001                     : ok=2    changed=2    unreachable=0    failed=0
```

As expected, Ansible successfully performed the MySQL dump of the Ansible database and notified the `nagios` server. Let's see if the check is updated on our `nagios` monitoring.

web001	mysql dump	**?**	OK	08-10-2014 11:48:20	0d 0h 0m 9s	1/1	OK: Dumped ansible mysql database

Hey, it worked! This was a simple example of Nagios integration to get you started.

Another example of Nagios alerting is to use it with some search indexing. For example, you perform an indexing of your site every midnight and want to monitor this using Nagios. To perform this activity, you can write a playbook, which will index your data and make sure that the production traffic is not affected because of this activity. At the end of the playbook, that is, the last task, you will notify your `nagios` server about the completion of the activity using a passive check. We'll leave this as an exercise for you to try out.

You can also add the `send_nsca` command to the Ansible callback and notify Nagios on each `failed`, `unreachable`, or `ok` status. Let's see how a callback script for this activity would look:

```
$ cat callback_nagios.py
import subprocess
class CallbackModule(object):
  def __init__(self):
    self.play_name = ""

  def nagios_passive_check(self, host, return_code, status):
    subprocess.call("echo -e '%s\t%s\t%d\t%s' | sudo /usr/sbin/send_nsca
-H 10.47.137.69  -c  /etc/nagios/send_nsca.cfg" % (host, self.play_name,
return_code, status),  shell=True)

  def runner_on_failed(self, host, res, ignore_errors=False):
    self.nagios_passive_check(host, 2, "Critical: %s" % res)

  def runner_on_unreachable(self, host, res):
    self.nagios_passive_check(host, 2, "Critical: %s" % res)

  def playbook_on_play_start(self, pattern):
    self.play_name = self.play.name

  def playbook_on_stats(self, stats):
    if not stats.dark and not stats.failures:
      self.nagios_passive_check(stats.ok.keys()[0], 0, "Ok: Successful")
```

Let's look at the flow of the code in more detail as follows:

1. We initialize a variable, `play_name`, with an empty value as follows. The intent is to store the playbook name in this variable so that we can access it from any of the methods within the `CallbackModule` class:

    ```
    def __init__(self):
      self.play_name = ""
    ```

2. We create a method to actually call the `send_nsca` command and update the `nagios` server as follows. We will pass the host against which Ansible is running, `return_code` for the `nagios` server, and finally the message, that is, the status of the playbook for the `nagios` server:

    ```
    def nagios_passive_check(self, host, return_code, status):
        subprocess.call("echo -e '%s\t%s\t%d\t%s' | sudo
            /usr/sbin/send_nsca -H 10.47.137.69  -c
                /etc/nagios/send_nsca.cfg" % (host,
                    self.play_name, return_code, status),
                        shell=True)
    ```

3. We assign the play name from the `playbook_on_stats` method. Make sure you add the correct play name to your playbook because we will be using this play name as a service name in nagios:

    ```
    def playbook_on_play_start(self, pattern):
        self.play_name = self.play.name
    ```

4. We call the `nagios_passive_check` method from the `runner_on_failed`, `runner_on_unreachable`, and `playbook_on_stats` methods. This method will notify the `nagios` server on receiving the `failure`, `unreachable`, and `ok` statuses, respectively. The `stats.dark` method in the third method, shown as follows, means the status was not `unreachable`, whereas `stats.failures` means the status was not `failure`:

    ```
    def runner_on_failed(self, host, res, ignore_errors=False):
        self.nagios_passive_check(host, 2, "Critical: %s" % res)


    def runner_on_unreachable(self, host, res):
        self.nagios_passive_check(host, 2, "Critical: %s" % res)


    def playbook_on_stats(self, stats):
        if not stats.dark and not stats.failures:
            self.nagios_passive_check(stats.ok.keys()[0], 0,
                "Ok: Successful")
    ```

Graphite

Graphite is yet another extensively used tool in operations for real-time graphing. Graphite thrives when it comes to time series. You can read more about Graphite at `http://graphite.readthedocs.org/en/latest/index.html`.

To use Graphite, you need to send the following three parameters to it:

- The metric name
- Value of the metric
- Timestamp

> Graphite provides an easy API that allows you to alert a user using a tool such as Nagios, Cabot, or Riemann. You can read more about Graphite at `http://graphite.wikidot.com`.

Now, the question is, what is it that we'd like to track from an Ansible perspective in Graphite? Typically, as a best practice from a configuration management perspective, we'd like to know how many tasks changed during an Ansible run. We'd also like to know whether the Ansible run itself passed or failed. If it failed, did it fail because the node was unreachable or because there was an issue with the playbook?

We'll do all of these from a Graphite plugin. We've seen how to check for status with Nagios. We'll go one step further by also monitoring the number of tasks and how many actually ran in each run. Let's see the code first, which is as follows:

```
$cat graphite_plugin.py

import socket
import time

class CallbackModule(object):

  def __init__(self):
    self.UNREACHABLE_RUN = 0
    self.FAILED_RUN = 25
    self.SUCCESSFUL_RUN = 50
    self.CARBON_SERVER = '192.168.1.3'
    self.CARBON_PORT = 2003
    self.playbook = ''

  #Process the run result for each host
  def process_result(self,res):
    status = self.SUCCESSFUL_RUN
```

```
        changed_tasks, tasks = 0,0
        if type(res) == type(dict()):
            for key in res:
                host = key
                if res[host]['unreachable'] == 1:
                    status = self.UNREACHABLE_RUN
                elif res[host]['failures'] != 0:
                    status = self.FAILED_RUN
                else:
                    tasks = res[host]['ok'] + res[host]['changed']
                    changed_tasks = res[host]['changed']
            host = host.replace('.','-')
            self.send_data_to_graphite(host,status,tasks,changed_tasks)


  def send_data_to_graphite(self, host, status, tasks, changed_tasks):
      prefix = "ansible.run.%s.%s" % (self.playbook,host)
      tasks_metric = "%s.number_of_tasks %d %d\n" %
(prefix,tasks,int(time.time()))
      status_metric = "%s.status %d %d\n" % (prefix,status,int(time.
time()))
      changed_tasks_metric = "%s.changed_tasks %d %d\n" % (prefix,changed_
tasks,int(time.time()))
      print "Prefix", prefix
      print "Tasks: %d, status: %d, changed_tasks: %s" %
(tasks,status,changed_tasks)


      sock = socket.socket()
      sock.connect((self.CARBON_SERVER, self.CARBON_PORT))
      sock.sendall(status_metric)
      sock.sendall(tasks_metric)
      sock.sendall(changed_tasks_metric)
      sock.close()

 ## Other methods in the plugin do not change  ##

 def playbook_on_play_start(self, pattern):
     self.playbook = pattern

 def playbook_on_stats(self, stats):
     results = dict([(h, stats.summarize(h)) for h in stats.processed])
     self.process_result(results)
```

Here, we see that we change only two methods in the callback:

- `playbook_on_play_start`: This method is used in order to store the name of the play
- `playbook_on_stats`: This method is used to get the final stats of the playbook run

The metric values that we'll send to Graphite are in the `send_data_to_graphite` method as follows:

- Metric for the overall number of tasks: (syntax: `ansible.run.<playbook name>.<hostname>.number_of_tasks`)

  To create a hierarchy in Graphite, you need to separate the metric names by using a dot. Hence, as part of this exercise, we replaced the dots in an IP address with a hyphen.

- Status of the run: (syntax: `ansible.run.<playbook name>.<hostname>. status`)

  As part of the status, we've covered three states: `SUCCESSFUL_RUN`, `FAILED_ RUN`, and `UNREACHABLE_RUN`. Each of the three statuses has specific values that you can easily map out in Graphite: `50`, `25`, and `0`. This is up to you to customize.

- Number of changed tasks: (syntax: `ansible.run.<playbook name>.<hostname>.changed_tasks`)

Make sure you have copied the graphite callback in the callbacks directory. We again use our `build_agent` role from the previous chapter to demonstrate this example. We bring up a new node and run the playbook. It results in the following output at the end of the run:

```
PLAY RECAP ***********************************************************************
Prefix ansible.run.build_agents.192-168-33-11
Tasks: 18, status: 50, changed_tasks: 8
192.168.33.11                    : ok=10    changed=8    unreachable=0    failed=0
```

Overall, 18 tasks ran and eight changed. Downloading of files does not change the state of the system and hence does not have the `changed` status associated with them.

Let's run these tasks again. We assume here that you run Ansible on each of the machines every hour, to maintain the state, or at the start of every build. We obviously haven't run the task every hour, because we want to show you what graphs are going to look like (We ran it within 5 minutes). So coming back, on rerunning, we find that eight tasks have run and none of the tasks have failed, as shown in the following screenshot:

```
PLAY RECAP ***********************************************************************
Prefix ansible.run.build_agents.192-168-33-11
Tasks: 8, status: 50, changed_tasks: 0
192.168.33.11                : ok=8     changed=0     unreachable=0     failed=0
```

You might be wondering how the jump from 18 to 8 happened. This is because the playbook has 10 tasks that depend on conditionals. For example, the task will download Maven if the directory is not present. Hence, all those tasks are skipped. You can look at the `build_agent` playbook role in the previous chapter to go over the playbook content again.

Time for an error

Now, we'll introduce an error in `ant.yml` and rerun the playbook. The output is shown as follows:

```
TASK: [build_agent | create ant directory] ************************************
failed: [192.168.33.11] => {"failed": true, "item": ""}
msg: this module requires key=value arguments (['dest/opt/', 'state=directory', 'owner=root', 'group=root'])

FATAL: all hosts have already failed -- aborting

PLAY RECAP ***********************************************************
{'unreachable': 0, 'skipped': 0, 'ok': 3, 'changed': 0, 'failures': 1}
Prefix ansible.run.build_agents.192-168-33-11
Tasks: 0, status: 25, changed_tasks: 0
          to retry, use: --limit @/Users/Madhurranjan/build_agent.retry

192.168.33.11                : ok=3     changed=0     unreachable=0     failed=1
```
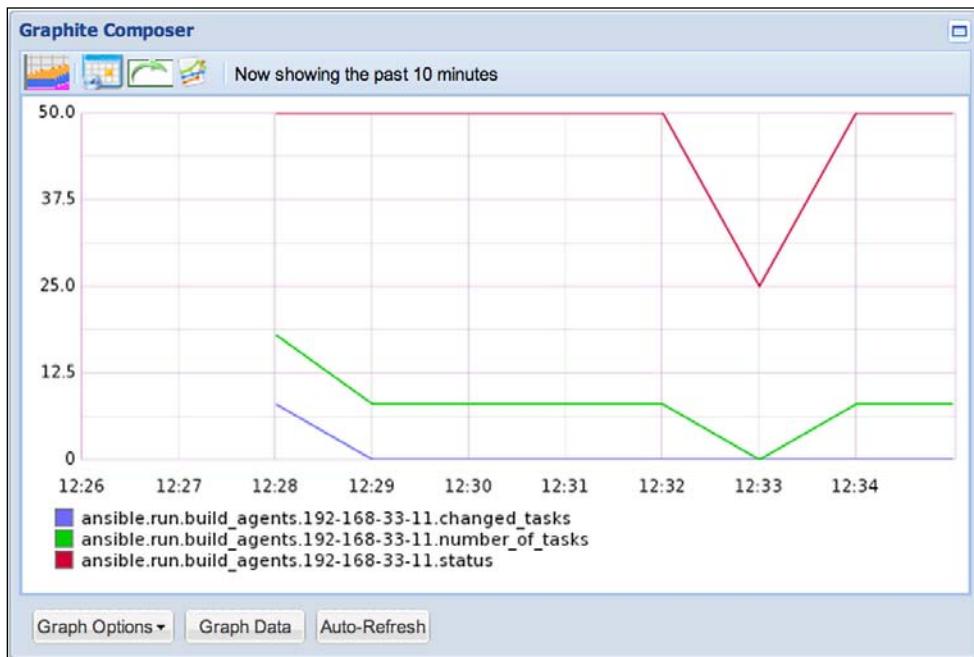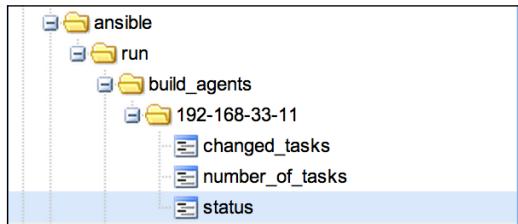
The Ansible run status will be set to FAILED_RUN from SUCCESSFUL_RUN and you will see a change in Graphite if everything works as expected.
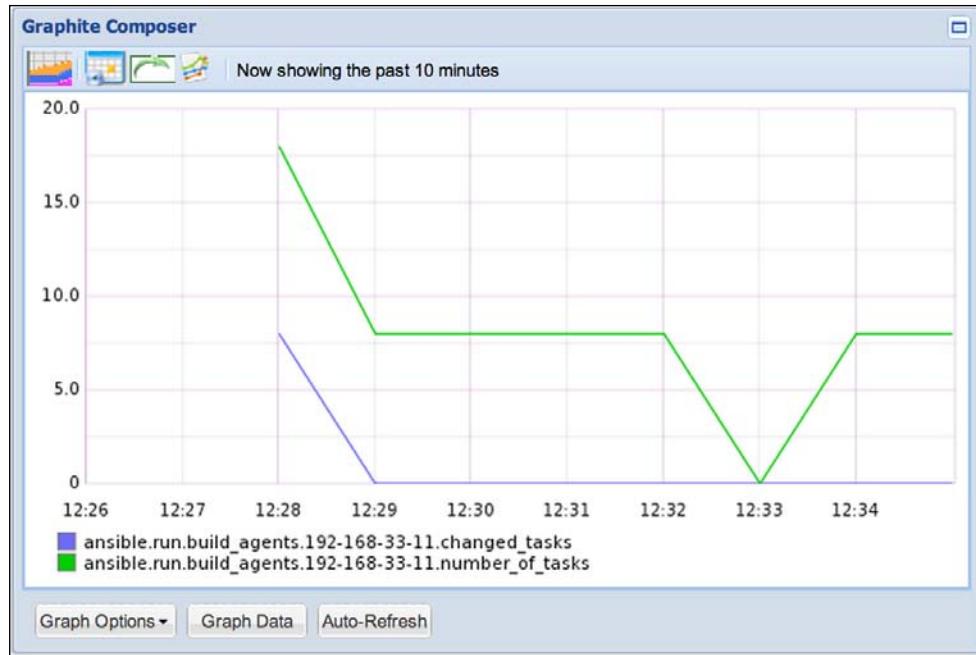
We then fix the error and rerun the playbook, and we again see eight tasks. Let's see the snapshots in Graphite spread across a 10-minute period that captures all these runs.

You can see the three metrics that we've chosen along with their respective values and how they've changed. On the left-hand side pane, you'll be able to see the hierarchy that we've introduced as follows:

Now, we remove the status to compare just the overall number of tasks and the number of changed tasks.

Over a period of time, if there is no change, you should have unchanged values for the `number_of_tasks` and `changed_tasks` metrics.

On the Graphite server filesystem, you will be able to see the following hierarchy for the three metrics that we measured:

```
[root@node build_agents]# pwd
/var/lib/carbon/whisper/ansible/run/build_agents
[root@node build_agents]# tree
.
└── 192-168-33-11
    ├── changed_tasks.wsp
    ├── number_of_tasks.wsp
    └── status.wsp

1 directory, 3 files
[root@node build_agents]# whisper-fetch 192-168-33-11/changed_tasks.wsp  | grep -v None
1407691680        8.000000
1407691740        0.000000
1407691860        0.000000
1407691980        0.000000
1407692040        0.000000
[root@node build_agents]#
```

You can see the directory from where we ran the `tree` command. Graphite stores all the data in the form of whisper files. You can query the values of each file on the filesystem using `whisper-fetch`, as shown in the preceding screenshot. You can also run a `curl` call to obtain the values in the JSON format. A `curl` call to obtain information about the last 30 minutes and in the JSON format should look like the following:

```
$ curl "http://<graphite ip>/render?target=ansible.run.build_agents.192-
168-33-11.*&format=json&from=-30min
```

The following is the output of the preceding command line:

The Graphite website has a great, detailed documentation on how to consume the data from Graphite. We'll leave it to you to figure out the rest! While consuming the Graphite metrics, let's say from Nagios, you can set up an alert that will filter out all the null values and alerts based on either the failure value or the lack of idempotency in your playbooks.

Summary

With this, we come to the end of this chapter where we've seen how to handle errors using rollbacks, how to use callbacks in general, and how to report and alert using tasks as well as callbacks. Along the way, we've also seen other tools such as `etcd`, MySQL database, HipChat, E-mail, Nagios, and Graphite, and how to integrate Ansible with them. These are definitely some common as well as easy implementations that provide great value to Ansible users. Finally, it's very important that you have a strong feedback loop in your infrastructure and you've seen a few of those techniques in action in this chapter.

It's also time for a coffee break. As usual, we have a few questions for you to ponder over and discuss with your team as follows:

- What monitoring system do you have and what features from this chapter would benefit you the most?
- If you use callbacks, what would you want your callbacks to do?
- If you want to alert, about what would you alert?

We hope that you'll revisit the roles that you've written, or started to write, after reading *Chapter 3*, *Taking Ansible to Production*, and incorporate some of the Ansible features that you've learned in this chapter to introduce feedback loops into your infrastructure automation. If you're a sysadmin, we can imagine you smiling after reading through this chapter!

The next chapter is all about writing custom modules and testing them. This will test some of your basic programming skills. So get ready to code.

5

Working with Custom Modules

"Customize your thoughts in order to personalize your behavior."

- Joseph Mercado, author of "The Undiscovered You"

The preceding quote is in the context of a human being's thoughts and behavior, but if you start to think about it and apply a similar philosophy to the technology and tools that you use, you'll feel that it fits in quite well.

This chapter will focus on how to write and test custom modules. We've already discussed how modules work and how to use them within your tasks. Well, just for a quick recap, a module in Ansible is a piece of code, which is transferred and executed on your remote host every time you run an Ansible task (it can also run locally if you've used `local_action`).

From our experience, we've seen custom modules being written whenever a certain functionality needs to be exposed as a first-class task. The same functionality could have been achieved without the module, but it would have required a series of tasks with existing modules to accomplish the end goal. For example, let's say you wanted to provision a server via **Preboot Execution Environment** (**PXE**). Without a custom module, you would have probably used a few shell or command tasks to accomplish the same. However, with a custom module, you would just pass the required parameters to it and the business logic will be embedded within the custom module in order to perform the PXE boot.

The arguments that you pass to a module, provided they are in a key-value format, will be forwarded in a separate file along with the module. Ansible expects at least two variables in your module output, (that is, the result of the module run) whether it passed or failed, and a message for the user, and they both have to be in the JSON format. If you adhere to this simple rule, you can customize as much as you want!

If you've already started implementing roles with what you had learned in *Chapter 3*, *Taking Ansible to Production*, we hope by the end of this chapter, you would revisit your playbooks and probably replace a few of your tasks as a single task by implementing a custom module. This way, you will constantly evolve your playbooks.

In this chapter, we will cover the following topics:

- Python modules
- Bash modules
- Ruby modules
- Testing modules

When you choose a particular technology or tool, you generally start with what it offers. You slowly understand the philosophy behind building the tool and what problems it helps you solve. However, you truly feel comfortable and in control only when you understand in depth how it works. At some stage, to utilize the complete power of a tool, you'll have to customize it in ways and means that suit your particular needs. Over a period of time, tools that provide you with an easy way to plug in new functionalities stay, and those that don't, disappear from the market.

It's a similar story with Ansible as well. All tasks in Ansible playbooks are modules of some kind and it comes loaded with hundreds of modules. You will find a module for almost everything you might need. However, there are always exceptions. This is where the power to extend it comes in. Chef provides **Lightweight Resources and Providers** (**LWRPs**) to perform this activity and Ansible allows you to extend its functionality using custom modules.

The significant difference, however, is that you can write the module in any language of your choice (provided you have an interpreter of that language), whereas in Chef, the module has to be in Ruby. We recommend using Python for any complex module, as there is out-of-the-box support to parse arguments; almost all `*nix` systems have Python installed by default and Ansible itself is written in Python. We will also show how you can write modules in other languages.

> To make your custom modules available to Ansible, you can either specify the path to your custom module in an environment variable, `ANSIBLE_LIBRARY`; use the `--module-path` command-line option, or drop the modules in a `./library` directory alongside your top-level playbooks.

With this background information, let's look at some code!

Using Python modules

Ansible intends to allow users to write modules in any language. Writing the module in Python, however, has its own advantages. You can take advantage of Ansible's libraries to shorten your code, an advantage not available for modules in other languages. Parsing user arguments, handling errors, and returning the required values becomes easier with the help of the Ansible libraries.

We will see two examples for a custom Python module, one with and one without using the Ansible library, to give you a glimpse of how custom modules work. Make sure you organize your directory structure as mentioned in the previous section before creating the module. The first example creates a module named `check_user`; let's look at the code in the following screenshot:

```
$ cat library/check_user
#!/usr/bin/env python

import pwd
import sys
import shlex
import json

def main():
    # Parsing argument file
    args = {}
    args_file = sys.argv[1]
    args_data = file(args_file).read()
    arguments = shlex.split(args_data)
    for arg in arguments:
        if "=" in arg:
            (key, value) = arg.split("=")
            args[key] = value
    user = args["user"]

    # Checking if user exists
    try:
        pwd.getpwnam(user)
        success = True
        ret_msg = "User %s exists" % user
    except KeyError:
        success = False
        ret_msg = "User %s does not exists" % user

    # Error handling and JSON return
    if success:
        print json.dumps({
            "msg": ret_msg
        })
        sys.exit(0)
    else:
        print json.dumps({
            "failed": True,
            "msg": ret_msg
        })
        sys.exit(1)


main()
```

The preceding custom module, `check_user`, will check whether a user exists on a host. The module expects a `user` argument from Ansible. Let's break down the preceding module and see what it does.

We first import the libraries required to parse the arguments. The arguments are shown in the following screenshot:

```
import pwd
import sys
import shlex
import json
```

Using the `sys` library, we then parse the arguments, which are passed in a file by Ansible. The arguments are in the format, `param1=value1 param2=value2`, where `param1` and `param2` are parameters and `value1` and `value2` are values of the parameters.

There are multiple ways to split arguments and create a dictionary and we've chosen an easy way to perform the operation. We first create a list of arguments by splitting the arguments with a whitespace character, and then separate the key and value by splitting the arguments with an `=` character and assigning it to a Python dictionary. For example, if you have a string such as `user=foo gid=1000`, then you will first create a list, which will look like `["user=foo", "gid=1000"]` and then loop over this list to create a dictionary. This dictionary will look like `{"user": "foo", "gid": 1000}`; this is shown in the following screenshot:

```
def main():
    # Parsing argument file
    args = {}
    args_file = sys.argv[1]
    args_data = file(args_file).read()
    arguments = shlex.split(args_data)
    for arg in arguments:
        if "=" in arg:
            (key, value) = arg.split("=")
            args[key] = value
    user = args["user"]
```

> We separate the arguments based on a whitespace character because this is followed by core Ansible modules. You can use any separator instead of a whitespace, but we would encourage you to maintain uniformity.

Once we have the user argument, we then check whether that user exists on the host as follows:

```
# Checking if user exists
try:
    pwd.getpwnam(user)
    success = True
    ret_msg = "User %s exists" % user
except KeyError:
    success = False
    ret_msg = "User %s does not exists" % user
```

We use the `pwd` library to check the `passwd` file for the user. For the sake of simplicity, we use two variables: one to store the success or failure message and the other to store the message for the user.

Finally, we use the variables created in the try-catch block to check if the module succeeded or failed, as shown in the following screenshot:

```
# Error handling and JSON return
if success:
    print json.dumps({
            "msg": ret_msg
    })
    sys.exit(0)
else:
    print json.dumps({
            "failed": True,
            "msg": ret_msg
    })
    sys.exit(1)
```

If the module succeeds, then we will exit the execution with an exit code `0`; else, we will exit with a `non-zero` code. Ansible will look for the `failed` variable and if it is set to `True`, it will exit unless you have explicitly asked Ansible to ignore errors using the `ignore_errors` parameter.

You can use customized modules like any other core module of Ansible. Let's look at an example playbook for the preceding custom module as follows:

```
$ cat playbooks/check_user.yml
---
-   hosts: web001
    remote_user: ec2-user
    gather_facts: no
    tasks:
      - name: Checking if user {{ user }} exists
        check_user: user={{ user }}
```

As you can see, we used the `check_user` module like any other core module. Ansible will execute this module on the remote host by copying the module to the remote host with arguments in a separate file. Let's see how this playbook runs as follows:

```
$ ansible-playbook -i hosts playbooks/check_user.yml -e "user=foo" -M library

PLAY [web001] ********************************************************

TASK: [Checking if user foo exists] *********************************
failed: [web001] => {"failed": true}
msg: User foo does not exists

FATAL: all hosts have already failed -- aborting

PLAY RECAP **********************************************************
           to retry, use: --limit @/Users/ramesh/check_user.retry

web001                     : ok=0    changed=0    unreachable=0    failed=1
```

As expected, since we do not have any existing user named `foo`, the module fails with a message saying `User foo does not exists`. If you noticed, we passed an extra option, `-M`, to tell Ansible where to look for custom modules.

Ansible also provides a Python library to parse user arguments and handle errors and returns. It's time to see how the Ansible Python library is useful to make your code shorter, faster, and less error prone. This is demonstrated in the following screenshot:

```
$ cat library/check_users
#!/usr/bin/env python

import pwd
import sys
import shlex
import json

def main():
    # Parsing argument file
    module = AnsibleModule(
        argument_spec = dict(
            user       = dict(required=True)
        )
    )
    user = module.params.get("user")

    # Checking if user exists
    try:
        pwd.getpwnam(user)
        success = True
        ret_msg = "User %s exists" % user
    except KeyError:
        success = False
        ret_msg = "User %s does not exists" % user

    # Error handling and JSON return
    if success:
        module.exit_json(msg=ret_msg)
    else:
        module.fail_json(msg=ret_msg)

from ansible.module_utils.basic import *
main()
```

Let's break down the preceding module and see how it works, as follows:

```
def main():
    # Parsing argument file
    module = AnsibleModule(
        argument_spec = dict(
            user       = dict(required=True)
        )
    )
    user = module.params.get("user")
```

Previously, we performed a lot of processing on the `argument` file to get the final user arguments. Ansible makes it easy by providing an `AnsibleModule` method, which does all the processing on its own and provides us with the final arguments. The `required=True` parameter means that the argument is mandatory and the execution will fail if the argument is not passed. The default value for `required` is `false`, which will allow users to skip the argument. You can then access the value of the arguments through the `module.params` dictionary by calling the `get` method on `module.params`.

The logic to check users on the remote host will remain the same, but the error handling and return aspect will change as follows:

```
# Error handling and JSON return
if success:
    module.exit_json(msg=ret_msg)
else:
    module.fail_json(msg=ret_msg)
```

Working with exit_json and fail_json

Ansible provides a shorter way to handle success and failure by providing the `exit_json` and `fail_json` methods, respectively. You can directly pass a message to these methods and Ansible will take care of the rest. You can also pass additional variables to these methods and Ansible will print those variables to `stdout`. For example, apart from the message, you might also want to print the `uid` and `gid` parameters of the user. You can do this by passing these variables to the `exit_json` method separated by a comma. Let's see how you can return multiple values to `stdout`, which is demonstrated in the following screenshot:

```
$ cat library/check_users_id
#!/usr/bin/env python

import pwd
import sys
import shlex
import json

class CheckUsers:
    def __init__(self, user):
        self.user = user

    def check_users(self):
        # Checking if user exists
        uid = ""
        gid = ""
        try:
            user = pwd.getpwnam(self.user)
            success = True
            ret_msg = "User %s exists" % self.user
            uid = user.pw_uid
            gid = user.pw_uid
        except KeyError:
            success = False
            ret_msg = "User %s does not exists" % self.user
        return success, ret_msg, uid, gid

def main():
    # Parsing argument file
    module = AnsibleModule(
        argument_spec = dict(
            user        = dict(required=True)
        )
    )
    user = module.params.get("user")

    chkusr = CheckUsers(user)
    success, ret_msg, uid, gid = chkusr.check_users()

    # Error handling and JSON return
    if success:
        module.exit_json(msg=ret_msg, uid=uid, gid=gid)
    else:
        module.fail_json(msg=ret_msg)


from ansible.module_utils.basic import *
main()
```

As you can see, we return the `uid` and `gid` of the user along with the message, `msg`. You can have multiple values and Ansible will print all of them in a dictionary format. You can also further use these values in your playbook by registering the output of the module (remember `register` in a playbook?).

Testing Python modules

You can test your module by simply running it along with an `arguments` file. Simply copy your arguments to a file and run it, as shown in the following screenshot:

```
$ cat arguments
user=foo
```

Now, pass the file over to your custom module, as shown in the following screenshot:

```
$ python library/check_user arguments
{"msg": "User foo does not exists", "failed": true}
```

Ansible provides a `test-module` script to test modules that use the Ansible library. To get this script, you should clone the Ansible repository at `https://github.com/ansible/ansible`. The `test-module` script is in the `hacking` folder within Ansible's `git` repository. You can now test your module, as shown in the following screenshot:

```
$ ansible/hacking/test-module -m library/check_user -a "user=foo"
* including generated source, if any, saving to: /Users/ramesh/.ansible_module_generated
* this may offset any line numbers in tracebacks/debuggers!
***********************************
RAW OUTPUT
{"msg": "User foo does not exists", "failed": true}


***********************************
PARSED OUTPUT
{
    "failed": true,
    "msg": "User foo does not exists"
}
```

In the preceding test, we pass the complete path of the module using the `-m` option and the arguments with the `-a` option. You can also pass multiple arguments to the module separated by a whitespace character.

> The syntax and behavior of the module will be the same if run from a playbook, no matter how you parse the arguments, handle errors, and return statuses from inside your module.

Using Bash modules

Bash modules in Ansible are no different than any other bash scripts, except the way it prints the data on `stdout`. Bash modules could be as simple as checking if a process is running on the remote host to running some complex commands.

> We recommend that you use bash over other languages, such as Python and Ruby only when you're performing simple tasks. In other cases, you should use languages that provide better error handling.

Let's see an example for the bash module as follows:

```
$ cat library/kill_java_process
#!/bin/bash
source $1

SERVICE=$service_name

JAVA_PIDS=$(/usr/java/default/bin/jps | grep $SERVICE| awk '{print $1}')

if [ $JAVA_PIDS ];then
    for JAVA_PID in $JAVA_PIDS
    do
        /usr/bin/kill -9 $JAVA_PID
    done
    echo "failed=False msg=\"Killed all the orphaned process for $SERVICE\""
    exit 0
else
    echo "failed=False msg=\"No orphaned process to kill for $SERVICE\""
    exit 0
fi
```

The preceding bash module will take the `service_name` argument and forcefully kill all of the Java processes that belong to that service. As you know, Ansible passes the argument file to the module. We then source the `arguments` file using `source $1`. This will actually set the environment variable with the name, `service_name`. We then access this variable using `$service_name` as follows:

```
source $1

SERVICE=$service_name
```

We then check to see if we obtained any PIDs for the service and run a loop over it to forcefully kill all of the Java processes that match `service_name`. Once they're killed, we exit the module with `failed=False` and a message with an exit code of `0`, as shown in the following screenshot:

```
if [ $JAVA_PIDS ];then
    for JAVA_PID in $JAVA_PIDS
    do
        /usr/bin/kill -9 $JAVA_PID
    done
    echo "failed=False msg=\"Killed all the orphaned process for $SERVICE\""
    exit 0
```

If we do not find any running process for the service, we will still exit the module with an exit code of 0 because terminating the Ansible run might not make sense; this is shown in the following screenshot:

```
else
    echo "failed=False msg=\"No orphaned process to kill for $SERVICE\""
    exit 0
fi
```

> You can still terminate the Ansible run by printing `failed=True` with an exit code of `1`

Ansible allows you to return a key-value output if the language itself doesn't support JSON. This makes Ansible more developer/sysadmin friendly and allows custom modules to be written in any language of one's choice.

Let's test the bash module by passing the `arguments` file to the module. The `arguments` file has the `service_name` parameter set to `Jenkins`, as shown in the following screenshot:

```
$ cat arguments
service_name=jenkins
```

Now, you can run the module like any other bash script. Let's see what happens when we do so in the following screenshot:

```
$ bash library/kill_java_process arguments
failed=False msg="No orphaned process to kill for jenkins"
```

As expected, the module did not fail even though there was no Jenkins process running on the localhost.

Using Ruby modules

Writing modules in Ruby is as easy as writing a module in Python or bash. You just need to take care of the arguments, errors, return statements, and of course, know basic Ruby! Let's see what a Ruby module looks like in the following screenshot:

```
$ cat library/rsync
#!/usr/bin/env ruby

require 'rsync'
require 'json'

src = ""
dest = ""
ret_msg = ""
success = ""

def print_message(state, msg, key="failed")
  message = {
    key    => state,
    "msg" => msg
  }
  print message.to_json
  exit 1 if state == false
  exit 0
end

args_file = ARGV[0]
data = File.read(args_file)
arguments = data.split(" ")
arguments.each do |argument|
  print_message(false, "Arguments should be name value pairs.Example: name=foo") if not argument.include?("=")
  field,value = argument.split("=")
  if field == "src"
    src = value
  elsif field == "dest"
    dest = value
  else print_message(false, "Invalid arguments provided . Valid arguments include src and dest")
  end
end

result = Rsync.run("#{src}", "#{dest}")
if result.success?
    success = true
    ret_msg = "Copied file successfully"
else
    success = false
    ret_msg = result.error
end

if success
  print_message(false, "#{ret_msg}")
else
  print_message(true, "#{ret_msg}")
end
```

In the preceding module, we first process the user arguments, then copy the
file using the `rsync` library, and finally, return the output. Let's break down
the preceding code and see how it works.

We first wrote a method, `print_message`, which will print the output in a JSON format. By doing this, we can reuse the same code in multiple places. Remember, the output of your module should contain `failed => true` if you want the Ansible run to fail; otherwise, Ansible will think that the module succeeded and will continue with the next task. The output obtained is as follows:

```
require 'rsync'
require 'json'

src = ""
dest = ""
ret_msg = ""
success = ""

def print_message(state, msg, key="failed")
  message = {
    key   => state,
    "msg" => msg
  }
  print message.to_json
  exit 1 if state == false
  exit 0
end
```

We then process the `arguments` file, which contains a key-value pair separated by a whitespace character. This is similar to what we did with the Python module earlier, where we took care of parsing out the arguments. We also perform some checks to make sure that the user has not missed any required argument.

In this case, we check if the `src` and `dest` parameters have been specified and print a message if the arguments are not provided. Further checks could include the format and type of arguments. You can add these checks and any other checks you deem important. For example, if one of your parameters is a date, then you'd like to verify that the input is actually the right date. Consider the following screenshot, which shows the discussed parameters:

```
args_file = ARGV[0]
data = File.read(args_file)
arguments = data.split(" ")
arguments.each do |argument|
  print_message(false, "Arguments should be name value pairs.Example: name=foo") if not argument.include?("=")
  field,value = argument.split("=")
  if field == "src"
    src = value
  elsif field == "dest"
    dest = value
  else print_message(false, "Invalid arguments provided . Valid arguments include src and dest")
  end
end
```

Once we have the required arguments, we will go ahead and copy the file using the `rsync` library as follows:

```
result = Rsync.run("#{src}", "#{dest}")
if result.success?
    success = true
    ret_msg = "Copied file successfully"
else
    success = false
    ret_msg = result.error
end
```

Finally, we check if the `rsync` task passed or failed and call the `print_message` function to print the output on `stdout` as follows:

```
if success
  print_message(false, "#{ret_msg}")
else
  print_message(true, "#{ret_msg}")
end
```

You can test your Ruby module by simply passing the `arguments` file to the module. Your `arguments` file will look as shown in the following screenshot:

```
$ cat arguments
src=/var/log/ansible.log   dest=/backup/ansible_149204.log
```

Let's now run the module, as shown in the following screenshot:

```
$ ruby rsync arguments
{"failed":false,"msg":"Copied file successfully"}$
```

We will leave the `serverspec` testing for you to complete.

Testing modules

Testing is often undervalued due to lack of understanding of its purpose and the benefits it can bring to the business. Testing modules is as important as testing any other part of the Ansible playbook because a small change in a module can break your entire playbook. We will take an example of the Python module that we wrote in the first section of this chapter and write an integration test using Python's nose test framework. Unit tests are also encouraged, but for our scenario where we check if a user exists remotely, an integration test makes more sense.

> **nose** is a Python test framework. For more information, visit
> `https://nose.readthedocs.org/en/latest/`.

To test the module, we convert our previous module into a Python class so that we can directly import the class in our test, and run only the main logic of the module. The following screenshot shows the restructured module, which will check whether a user exists on a remote host:

```
$ cat library/check_users
#!/usr/bin/env python

import pwd
import sys
import shlex
import json

class User:
    def __init__(self, user):
        self.user = user

    def check_if_user_exists(self):
        # Checking if user exists
        try:
            pwd.getpwnam(self.user)
            success = True
            ret_msg = "User %s exists" % self.user
        except KeyError:
            success = False
            ret_msg = "User %s does not exists" % self.user
        return success, ret_msg

def main():
    # Parsing argument file
    module = AnsibleModule(
        argument_spec = dict(
            user        = dict(required=True)
        )
    )
    user = module.params.get("user")

    chkusr = User(user)
    success, ret_msg = chkusr.check_if_user_exists()

    # Error handling and JSON return
    if success:
        module.exit_json(msg=ret_msg)
    else:
        module.fail_json(msg=ret_msg)


from ansible.module_utils.basic import *
if __name__ == "__main__":
    main()
```

As you can see in the preceding screenshot, we created a class named `User`. We instantiated the class, and called the `check_if_user_exists` method to check if the user actually exists on the remote machine.

It's time to write an integration test now. We assume that you have the `nose` package installed on your system. If not, don't worry! You can still install the package by using the following command:

pip install nose

Let's now write the integration test as follows:

```
$ cat test_check_users.py
from nose.tools import *
import imp
imp.load_source("check_users","./check_users")
from check_users import User

def test_check_user():
    chkusr = User("ec2-user")
    success, ret_msg = chkusr.check_if_user_exists()
    assert_true(success)
    assert_equals('User ec2-user exists', ret_msg)
```

In the preceding integration test, we import the `nose` package and our module, `check_users`. We call the `User` class by passing the user we want to check. We then check whether the user exists on the remote host by calling the `check_if_user_exists()` method. Nose methods, `assert_true` and `assert_equals`, can be used to compare the expected value against the actual. Only if the `assert` methods pass, will the test pass. You can have multiple tests inside the same file by having multiple methods whose names start with `test_`, for example, the `test_if_user_exists()` method. Nose tests will take all the methods that start with `test_` and execute them. Let's see how this works in the following screenshot:

```
$ nosetests -v test_check_users.py
library.test_check_users.test_check_user ... ok

----------------------------------------------------------------------
Ran 1 test in 0.002s

OK
```

As you can see, the test passed because the user, `ec2-user`, existed on the host. We use the `-v` option with nose tests for the verbose mode. For more complicated modules, we recommend that you write unit tests and integration tests. You might wonder why we didn't use `serverspec` to test the module.

We still recommend running `serverspec` tests for functional testing as part of playbooks, but for unit and integration tests, it's recommended to use well-known frameworks.

Finally, we recommend that your run all these tests as part of your CI system, be it `Jenkins`, `Travis`, or any other system.

Similarly, if you write Ruby modules, we recommend you write tests for them with a framework such as `rspec`. If your custom Ansible module has multiple parameters with multiple combinations, then you will write more tests to test each scenario. Finally, we recommend that you run all these tests as part of your CI system, be it `Jenkins`, `Travis`, or any other system.

Summary

With this, we come to the end of this rather small but important chapter, which focused on how you can extend Ansible by writing your own custom modules. You learned how to use Python, Bash, and Ruby in order to write your modules. We've also seen how to write integration tests for modules so that they can be integrated into your CI system. In future, hopefully, extending your Ansible functionality using modules should be way easier!

A couple of questions to think about are as follows:

- Can you think of common tasks that you perform daily and how you would write an Ansible module for that? List them down in terms of how you would invoke the module from a playbook.

- Which language do you think your team would be comfortable using for your modules?

- Can you revisit the roles that you might have written after *Chapter 3*, *Taking Ansible to Production*, and see which of them can potentially be converted into custom modules?

Next, we will step into the world of Provisioning, Deployment, and Orchestration and look at how Ansible solves our infrastructure problems when we provision new instances or want to deploy software updates to various instances in our environments. We promise that the journey is going to be fun!

6
Provisioning

"The only kind of server you want to create is a Phoenix server"

– Madhurranjan and Ramesh

We've now hit the "slog overs" as they say in cricket or the part of the book from where we can see a logical end to the book. However, before we do that, we have a few topics that anyone in System Administration, Release Management, or DevOps would love to discuss—Provisioning, Deployment, and Orchestration. We're going to cover topics on Deployment and Orchestration in the next chapter, and all that we think is important regarding Provisioning in this chapter.

Regarding the preceding quote, the term "Phoenix Server" first appeared in one of Martin Fowler's famous blikis (`http://martinfowler.com/bliki/PhoenixServer. html`) and the credit goes to his colleague *Kornelis Sietsma* for coining the term. The concept of a Phoenix is quite popular. For those who haven't heard, it refers to a Greek mythological bird that lived long and cyclically regenerated from its own ashes. Harry Potter fans might have come across the bird either in the books or movies. The end result is that, while designing your infrastructure, you want to make sure that you can bring up and bring down machines on demand and not allow certain machines to assume mammoth importance.

One of the authors was in charge of a decently large build system and he came across these finely carved custom build machines (Windows VMs to be specific) that had been running builds for close to 3 years. Their team had to relocate close to 200 such machines from one data center to another, since they were shutting down the first data center. The team spent close to a month and the simple algorithm they followed was to shut down machines once everyone went home, copy as many VM images as possible onto a USB drive during the night, and physically drop the drive at the data center the next morning so that someone in charge could mount them and copy all the images over to the right servers. If you have an exasperated look on your face, it's quite understandable because the entire team felt quite stupid during the entire month. One of the immediate projects that was commissioned after that was to make sure that they evolved to Phoenix servers soon after that.

In this chapter, at a broad level, we'll cover the following topics:

- Provisioning of machines in cloud such as AWS and DigitalOcean
- Provisioning Docker containers
- Ansible's Dynamic Inventory

Most of the new machine creations have two phases:

- Provisioning a new machine
- Running scripts to make sure the machine is configured to play the right role in your infrastructure

We've looked at the configuration management aspect in the initial chapters. We'll focus a lot more on provisioning new machines in this chapter with a lesser focus on configuration management.

Provisioning a machine in the cloud

With that, let's jump to the first topic. Teams managing infrastructures have a lot of choices today to run their builds, tests, and deployments. Providers such as Amazon, Rackspace, and DigitalOcean primarily provide Infrastructure as a Service (IAAS). They expose an API via SDKs, which you can invoke in order to create new machines, or use their GUI to set it up. We're more interested in using their SDK as it will play an important part in our automation effort. Setting up new servers and provisioning them is interesting at first but at some stage it can become boring as it's quite repetitive in nature. Each provisioning step would involve several similar steps to get them up-and-running.

Imagine one fine morning you receive an e-mail asking for three new customer setups, where each customer setup has three to four instances and a bunch of services and dependencies. This might be an easy task for you, but would require running the same set of repetitive commands multiple times, followed by monitoring the servers once they come up to confirm that everything just went fine. In addition, anything you do manually has a chance of introducing bugs.

What if two of the customer setups come up correctly but, due to fatigue, you miss out a step for the third customer and hence introduce a bug? To deal with such situations, there exists automation. Cloud provisioning automation makes it easy for an engineer to build up a new server as quickly as possible, allowing him/her to concentrate on other priorities. Using Ansible, you can easily perform these actions and automate cloud provisioning with minimal effort. Ansible provides you with the power to automate various different cloud platforms, such as Amazon, DigitalOcean, Google Cloud, Rackspace, and so on, with modules for different services available in the Ansible core.

In this section, we will see two examples of cloud provisioning:

- One for Amazon
- One for DigitalOcean

> As mentioned earlier, bringing up new machines is not the end of the game. We also need to make sure we configure them to play the required role. Let's look at a popular use case—that of bringing up a Hadoop cluster to run MapReduce jobs. In short, for the uninitiated, Hadoop is a framework that supports storage and large scale processing of datasets on clusters of commodity hardware. More information regarding Hadoop can be found at `http://hadoop.apache.org/`.

The directory structure looks as follows:

```
$ tree
.
├── hosts
└── playbooks
    ├── cloud_provision.yml
    ├── digital_ocean_launch.yml
    └── ec2_launch.yml

1 directory, 4 files
```

There are three playbooks:

- One to launch an Amazon instance
- One to launch a DigitalOcean instance
- One common playbook to install the Hadoop cluster

We also have a static inventory file that consists of only one host, the localhost. This file can either be /etc/ansible/hosts or any other file that needs to be referred to with the -i option while running ansible-playbook. You might be wondering why the host file just has localhost. This is because we'll be launching the machines from the localhost. In this case, localhost is also the command center from where we execute playbooks.

```
$ cat hosts
[local]
localhost
```

We will modify this inventory file on-the-fly, dynamically from the playbook with the new host details. Let's see how we can use Ansible to launch a couple of ec2 instances and install the Hadoop cluster on it:

```
$ cat playbooks/ec2_launch.yml
- name: Provision hadoop name node
  local_action: ec2  keypair={{ mykeypair }} instance_type={{ instance_type }} image={{ image }} wait=
true zone={{ zone }} region={{ region }} instance_tags='{"name":"{{ item }}"}' wait_timeout=600
  register: ec2
  with_items:
    - "hadoop-name"
    - "hadoop-data"

- name: Print new instance details
  debug: var=ec2

- name: Wait for SSH to come up
  local_action: wait_for host={{ item.instances[0]['public_dns_name'] }} port=22 delay=60 timeout=320
state=started
  with_items: ec2.results

- name: Add new instance to host group
  add_host: hostname={{ item.instances[0]['public_ip'] }} groupname=launched instance_hostname={{ item
.item }} user=ec2-user
  with_items: ec2.results
```

Diving deep into the playbook

In the first task shown in the preceding screenshot, we launch the `ec2` instance with the `hadoop-name` and `hadoop-data` tags, respectively, so that we can later identify the new instances easily. We register the output of the `ec2` module to a variable called `ec2`, which can be used to get the new instance details in the later tasks:

```
- name: Provision hadoop name node
  local_action: ec2  keypair={{ mykeypair }} instance_type={{ instance_
type }} image={{ image }} wait=true zone={{ zone }} region={{ region }}
instance_tags='{"name":"{{ item }}"}' wait_timeout=600
  register: ec2
  with_items:
    - "hadoop-name"
    - "hadoop-data"
```

The `with_items` section in the preceding code is an array that consists of tags for the new machines that are going to be provisioned. The way `item` is referenced is shown in the following line of code:

```
instance_tags='{"name":"{{ item }}"}'
```

The `item` section takes the value `hadoop-name` in the first iteration for the first machine's tag and `hadoop-data` in the next iteration for the second machine's tag.

The next task will print the new instance details that we just launched:
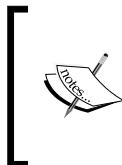
```
- name: Print new instance details
  debug: var=ec2
```

We then wait for the SSH port to come up using the `wait_for` module. You can use this module to wait for a specific port to come up. You can also wait until a specific file is present or absent or a specific regular expression is matched. The `with_items` section here provides the two instances that were created in the earlier step and registered with the `ec2` variable as parameters. The `item.instances[0]['public_dns_name']` represents `public_dns_name` for each of the instances that we provisioned and is provided to the `host` parameter as part of the `wait_for` check:

```
- name: Wait for SSH to come up
  local_action: wait_for host={{ item.instances[0]['public_dns_name'] }}
port=22 delay=60 timeout=320 state=started
  with_items: ec2.results
```

The last task will update the inventory file with a new group called `launched` and add the two newly launched hosts, provided as part of the `with_items` construct, under it using the `add_host` module. This group will be valid only for the current Ansible run, that is, the module will not actually update the inventory file; instead, it will load that group in the memory for the current Ansible run. It will also add two variables, `instance_hostname` and `user`, in the inventory against the new host, which we can use later in the playbook to identify the hostname and SSH user, respectively, as follows:

```
- name: Add new instance to host group
  add_host: hostname={{ item.instances[0]['public_ip'] }}
groupname=launched instance_hostname={{ item.item }} user=ec2-user
  with_items: ec2.results
```

> To launch an instance in `ec2`, you either need to set the environment variables, `AWS_ACCESS_KEY` and `AWS_SECRET_KEY`, for the AWS access key and the secret key, respectively, or pass it to the `ec2` module in the preceding playbook. You can generate these keys from your AWS console.

We will now include this playbook in the `cloud_provision.yml` playbook. The `cloud_provision.yml` playbook allows us to launch an instance(s) either in Amazon EC2 or in DigitalOcean and then installs the required packages and dependencies to configure the Hadoop cluster. Let's now look at the playbook by running the following command:

```
$ cat playbooks/cloud_provision.yml
```

The following command lines denote play 1:

```
---
- name: Launching new hosts
  hosts: localhost
  gather_facts: no
  tasks:
    - include: digital_ocean_launch.yml tags=digital_ocean
    - include: ec2_launch.yml tags=ec2
```

The following command lines denote play 2:

```
- name: Updating new instance
  hosts: launched
  gather_facts: no
  user: "{{ user }}"
  sudo: yes
  tags: digital_ocean,ec2
  tasks:
    - name: Updating hostname
      hostname: name={{ instance_hostname }}

    - name: Installing wget
      yum: name=wget state=present

    - name: Grouping hosts
      group_by: key={{ instance_hostname }}

    - name: Downloading CDH Repository
      shell: wget http://archive.cloudera.com/cdh4/one-click-install/
redhat/6/x86_64/cloudera-cdh-4-0.x86_64.rpm

    - name: Installing CDH Repository
      yum: name=cloudera-cdh-4-0.x86_64.rpm state=installed

    - name: Adding Cloudera Public GPG Key to repository
      shell: rpm --import http://archive.cloudera.com/cdh4/redhat/6/
x86_64/cdh/RPM-GPG-KEY-cloudera
```

The following command lines denote play 3:

```
- name: Provisioning name node
  hosts: hadoop-name*
  user: "{{ user }}"
  sudo: yes
  tags: digital_ocean,ec2

  tasks:
```

```
   - name: Installing required packages

     yum: name=hadoop-0.20-mapreduce-jobtracker,hadoop-hdfs-
namenode,hadoop-hdfs-secondarynamenode state=present


   - name: starting services
     service: name={{ item }} state=started
     register: ser
     failed_when: "'Starting' not in ser.msg"
     with_items:
        - hadoop-0.20-mapreduce-jobtracker
        - hadoop-hdfs-namenode
        - hadoop-hdfs-secondarynamenode
```

The following command lines denote play 4:

```
- name: Provisioning  data node
  hosts: hadoop-data*
  user: "{{ user }}"
  sudo: yes
  tags: digital_ocean,ec2
  tasks:
    - name: Installing required packages

      yum: name=hadoop-0.20-mapreduce-tasktracker,hadoop-hdfs-
datanode,hadoop-client state=present


    - name: starting services
      service: name={{ item }} state=started
      register: ser
      with_items:
         - hadoop-0.20-mapreduce-tasktracker
         - hadoop-hdfs-datanode
      failed_when: "'Starting' not in ser.msg"
```

As shown in the preceding snippet of command lines, `cloud_provision.yml` is the common playbook, which will be used for both AWS and DigitalOcean. This playbook consists of four different plays. At a high level, the following are the plays:

- **The first play**: This launches new nodes either in AWS or Digital Ocean
- **The second play**: This updates the provisioned nodes with common content
- **The third play**: This is specific to NameNode
- **The fourth play**: This is specific to DataNode

Let's examine each of the plays in more detail.

The first play will launch an instance in `ec2` or DigitalOcean depending on the tags passed. If the tag passed is `ec2`, it will kick off an `ec2` deployment by calling `ec2_launch.yml`; alternatively, if the tag is `digital_ocean`, it will kick off a DigitalOcean deployment by calling `digital_ocean_launch.yml`:

```
- name: Launching new hosts
  hosts: localhost
  gather_facts: no

  tasks:

    - include: digital_ocean_launch.yml tags=digital_ocean

    - include: ec2_launch.yml tags=ec2
```

The second play will set the hostname on the new hosts and install other dependencies:

```
- name: Updating new instance
  hosts: launched
  gather_facts: no
  user: "{{ user }}"
  sudo: yes
  tags: digital_ocean,ec2

  tasks:
    - name: Updating hostname
      hostname: name={{ instance_hostname }}

    - name: Installing wget
      yum: name=wget state=present

    - name: Grouping hosts
      group_by: key={{ instance_hostname }}

    - name: Downloading CDH Repository
      shell: wget http://archive.cloudera.com/cdh4/one-click-install/redhat/6/x86_64/cloudera-cdh-4-0.x86_64.rpm

    - name: Installing CDH Repository
      yum: name=cloudera-cdh-4-0.x86_64.rpm state=installed

    - name: Adding Cloudera Public GPG Key to repository
      shell: rpm --import http://archive.cloudera.com/cdh4/redhat/6/x86_64/cdh/RPM-GPG-KEY-cloudera
```

In the second play, we used hosts in the `launched` group of the inventory, which we added in the previous play using `hosts: launched`.

The first task in the second play updates the hostname on the new instances using the `hostname` module. The `instance_hostname` variable is added to the inventory file in the preceding play:

```
- name: Updating hostname
  hostname: name={{ instance_hostname }}
```

The following tasks will install dependencies and the repository to install a Hadoop cluster:

```
- name: Installing wget
  yum: name=wget state=present


- name: Downloading CDH Repository
  shell: wget http://archive.cloudera.com/cdh4/one-click-install/
redhat/6/x86_64/cloudera-cdh-4-0.x86_64.rpm


- name: Grouping hosts  ---- ( We'll look at this task next!)
  group_by: key={{ instance_hostname }}


- name: Installing CDH Repository
  yum: name=cloudera-cdh-4-0.x86_64.rpm state=installed


- name: Adding Cloudera Public GPG Key to repository
  shell: rpm --import http://archive.cloudera.com/cdh4/redhat/6/
x86_64/cdh/RPM-GPG-KEY-cloudera
```

Let's look at the `group_by` module. The `group_by` module allows you to split hosts further based on certain parameters, be it hostnames, machine facts, or even regular expressions.

In this case, the `group_by` task will further group the instances based on the hostname inside the `launched` group. Let's consider, for example, that the `launched` group contains the hosts hdpn001, hdpd002, hdpd003, and hdpd004. Out of these, hdpn001, as the name suggests, is the Hadoop NameNode whereas the other three hosts are Hadoop DataNodes. We can now group these hosts based on either the hostname or any other Ansible fact. If you are grouping the hosts based on the hostname, then you can simply use regex inside your `hosts` field, as follows:

```
hosts:  hdpd*
```

The preceding regex will match all hosts that start with `hdpd`; these are Hadoop DataNodes in our case. Another popular use case is when you're running it across a bunch of database nodes, where certain nodes are masters and certain nodes are slaves. You can run `group_by` on a `db_role` fact that is set to either `master` or `slave` and create a subgroup on which you can run certain actions.

Ansible again maintains the group only until the current Ansible run and it is not reflected in the actual inventory file. Using these groups, we can easily identify our `name` and `data` nodes in further plays. For this particular case, we create groups here to separate out the nodes as DataNodes and NameNodes:

```
- name: Grouping hosts
  group_by: key={{ instance_hostname }}
```

The last two plays are specific to NameNode and DataNode, respectively:

```
- name: Provisioning name node
  hosts: hadoop-name*
  user: "{{ user }}"
  sudo: yes
  tags: digital_ocean,ec2

  tasks:
    - name: Installing required packages
      yum: name=hadoop-0.20-mapreduce-jobtracker,hadoop-hdfs-namenode,hadoop-hdfs-secondarynamenode state=present

- name: Provisioning  data node
  hosts: hadoop-data*
  user: "{{ user }}"
  sudo: yes
  tags: digital_ocean,ec2

  tasks:

    - name: Installing required packages
      yum: name=hadoop-0.20-mapreduce-tasktracker,hadoop-hdfs-datanode,hadoop-client state=present
```

We use regex in the `hosts` field to map the names of hosts. This regex will identify hosts that have matching hostnames:

```
hosts: hadoop-name*
```

The preceding regex will identify all hosts that start with the string `hadoop-name`. This means that hosts with hostnames `hadoop-name01`, `hadoop-name02`, `hadoop-name-test`, and so on will be matched. These groups were created in play 2 in the preceding screenshot using the `group_by` module.

Provisioning

Let's run the preceding playbook to launch two instances on the Amazon EC2 cloud and install the Hadoop cluster on it. We call the `cloud_provision.yml` playbook with the following extended parameters:

- `Group_id`: This is a security group in AWS
- `Mykeypair`: This is a key pair generated in AWS
- `Instance_type`: This is a type of instance that will be provisioned
- `Image id`: This is an **Amazon Machine Image** (**AMI**), which is an image OS template used to spawn the new machine
- `zone`: This specifies the AWS zone
- `region`: This specifies the AWS region

We also provide –tags=ec2 so that the `ec2_launch.yml` file is invoked, as follows:

```
$ ansible-playbook -i hosts playbooks/cloud_provision.yml -e "group_
id=sg-bddb8bd6 mykeypair=ramesh-opsresearch instance_type=t1.micro
image=ami-bba18dd2 zone=us-east-1b region=us-east-1" --tags ec2


PLAY [Launching new hosts] *********************************************
******


TASK: [Provision hadoop name node] ************************************
******
changed: [localhost] => (item=hadoop-name)
changed: [localhost] => (item=hadoop-data)


TASK: [Print new instance details] ************************************
******
ok: [localhost] => {
    "ec2": {
        "changed": true,
        "msg": "All items completed",
        "results": [
            {
                "changed": true,
                "instance_ids": [
                    "i-98458573"
                ],
```

```
        "instances": [
            {
                "ami_launch_index": "0",
                "architecture": "x86_64",
                "dns_name": "ec2-54-198-250-211.compute-
                    1.amazonaws.com",
                "ebs_optimized": false,
                "hypervisor": "xen",
                "id": "i-43be45a8",
                "image_id": "ami-bba18dd2",
                "instance_type": "t1.micro",
                "kernel": "aki-919dcaf8",
                "key_name": "ramesh-opsresearch",
                "launch_time": "2014-08-31T16:24:05.000Z",
                "placement": "us-east-1b",
                "private_dns_name": "ip-10-71-130-
                    46.ec2.internal",
                "private_ip": "10.71.130.46",
                "public_dns_name": "ec2-54-198-250-
                    211.compute-1.amazonaws.com",
                "public_ip": "54.198.250.211",
                "ramdisk": null,
                "region": "us-east-1",
                "root_device_name": "/dev/sda1",
                "root_device_type": "ebs",
                "state": "running",
                "state_code": 16,
                "virtualization_type": "paravirtual"
            }
        ],
```

The output is truncated because it's way too long. In the very first task, you can see that we launched two instances and we see a changed status twice because we ran a loop for `hadoop-name` and `hadoop-datanode`; that is, two nodes were created (remember how it was with items?). Let's see how the other three plays behaved:

```
PLAY [Updating new instance] **********************************************

TASK: [Updating hostname] *************************************************
changed: [54.227.43.141]
changed: [54.198.250.211]

TASK: [Installing wget] ***************************************************
ok: [54.227.43.141]
ok: [54.198.250.211]

TASK: [Grouping hosts] ****************************************************
changed: [54.198.250.211]

TASK: [Downloading CDH Repository] ****************************************
changed: [54.227.43.141]
changed: [54.198.250.211]

TASK: [Installing CDH Repository] *****************************************
changed: [54.227.43.141]
changed: [54.198.250.211]

TASK: [Adding Cloudera Public GPG Key to repository] **********************
changed: [54.227.43.141]
changed: [54.198.250.211]

PLAY [Provisioning name node] *********************************************

GATHERING FACTS ***********************************************************
ok: [54.198.250.211]

TASK: [Installing required packages] **************************************
changed: [54.198.250.211]

PLAY [Provisioning  data node] ********************************************

GATHERING FACTS ***********************************************************
ok: [54.227.43.141]

TASK: [Installing required packages] **************************************
changed: [54.227.43.141]

PLAY RECAP ****************************************************************
54.198.250.211             : ok=8    changed=6    unreachable=0    failed=0
54.227.43.141              : ok=8    changed=6    unreachable=0    failed=0
localhost                  : ok=4    changed=1    unreachable=0    failed=0
```

The preceding screenshot is the output of the playbook that we ran previously. As you can see, Ansible ran all four plays that launched two `ec2` instances and installed a Hadoop NameNode and DataNode. We have not added any task to customize the Hadoop cluster and will leave that up to you as an exercise, based on your setup.

Launching a DigitalOcean instance
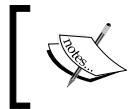
Let's now see how you can launch a DigitalOcean instance using Ansible. We will use the `digital_ocean_launch.yml` playbook to launch an instance and use the same `cloud_provision.yml` playbook to install dependencies and the Hadoop cluster that we used for `ec2`. Let's briefly look at the `digital_ocean_launch` playbook:

```
$ cat playbooks/digital_ocean_launch.yml
- name: Provision hadoop name node
  local_action: digital_ocean state=present ssh_key_ids=261020 name={{ item }} api_key=zdkwd589cf22b86fcc1e94d14d5011 image_id=344
8641 size_id=66 region_id=8 client_id=uywdkla4a6875a3f6c0695cfc8001277
  register: droplet
  with_items:
    - "hadoop-name"
    - "hadoop-data"

- debug: var=droplet

- name: Add new instance to host group
  add_host: hostname={{ item.droplet.ip_address }} groupname=launched instance_hostname={{ item.item }} user=root
  with_items: droplet.results
```

This playbook will launch two instances in the DigitalOcean cloud and update the inventory accordingly.

> To launch an instance, you will need to generate an API key and client ID from the DigitalOcean console and pass it to the `digital_ocean` module in the preceding playbook.

We included this playbook in the `cloud_provision.yml` playbook and can now run it with a `digital_ocean` tag. Please remember that, in this case, the playbook that we call does not change; just the tags and extra arguments to the playbook change.

Let's run this playbook and see how it behaves:

```
$ ansible-playbook -i hosts playbooks/cloud_provision.yml --tags digital_ocean

PLAY [Launching new hosts] ****************************************************

TASK: [Provision hadoop name node] *******************************************
changed: [localhost] => (item=hadoop-name)
changed: [localhost] => (item=hadoop-data)

TASK: [debug var=droplet] ****************************************************
ok: [localhost] => {
    "droplet": {
        "changed": true,
        "msg": "All items completed",
        "results": [
            {
                "changed": true,
                "droplet": {
                    "backups": [],
                    "backups_active": false,
                    "created_at": "2014-08-31T17:57:11Z",
                    "event_id": 31710472,
                    "id": 2504823,
                    "image_id": 3448641,
                    "ip_address": "104.131.22.248",
                    "locked": false,
                    "name": "hadoop-name",
                    "private_ip_address": null,
                    "region_id": 8,
                    "size_id": 66,
                    "snapshots": [],
                    "status": "active"
                },
```

The output is truncated, but you can see, Ansible launched both the instances and printed the output with the instance details. Let's see what happens to the Hadoop installation:

```
PLAY [Updating new instance] ****************************************************

TASK: [Updating hostname] ******************************************************
ok: [104.131.22.249]
ok: [104.131.22.248]

TASK: [Installing wget] ********************************************************
changed: [104.131.22.248]
changed: [104.131.22.249]

TASK: [Grouping hosts] *********************************************************
changed: [104.131.22.248]

TASK: [Downloading CDH Repository] *********************************************
changed: [104.131.22.248]
changed: [104.131.22.249]

TASK: [Installing CDH Repository] **********************************************
changed: [104.131.22.248]
changed: [104.131.22.249]

TASK: [Adding Cloudera Public GPG Key to repository] ***************************
changed: [104.131.22.248]
changed: [104.131.22.249]

PLAY [Provisioning name node] **************************************************

GATHERING FACTS ****************************************************************
ok: [104.131.22.248]

TASK: [Installing required packages] *******************************************
changed: [104.131.22.248]

PLAY [Provisioning  data node] *************************************************

GATHERING FACTS ****************************************************************
ok: [104.131.22.249]

TASK: [Installing required packages] *******************************************
changed: [104.131.22.249]

PLAY RECAP *********************************************************************
104.131.22.248             : ok=8    changed=6    unreachable=0    failed=0
104.131.22.249             : ok=8    changed=6    unreachable=0    failed=0
localhost                  : ok=3    changed=1    unreachable=0    failed=0
```

There you go! You have an up-and-running Hadoop cluster that is fully functional. We've shown how you can use Ansible to provision new machines in AWS and DigitalOcean; add them to the inventory; and, finally, configure them so that they end up performing the activity they're meant for. The methodology that you will follow while dealing with any other cloud or your own data center will be very similar to what we've shown if you're using Ansible or any other similar tool; hopefully, we've shown enough to get you started and be productive.

Docker provisioning

Docker is perhaps the most popular open source tool that has been released in the last year. The following quote can be seen on the Docker website:

> *Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications, whether on laptops, data center VMs, or the cloud.*

Increasingly, more and more individuals and companies are adopting Docker. The tagline for Docker is *Containerization is the new virtualization*. At a high level, all Docker allows you to do is prepare lightweight containers using instructions from a Dockerfile and run the container. The same container can be shared or shipped across environments, thereby making sure you run the exact same image and reducing the chance of errors. The Docker image that you build is cached by default; thus, the next time you have similar instructions, the time taken to bring up a similar container is reduced to almost nothing.

What is a container?

Though this is not a book about containers, we've commonly heard concerns and questions regarding what a container can provide as against a VM. We'd like to devote a paragraph to what a Linux container is before proceeding; this is so that we can see how Ansible can aid your use of Docker to make it a simple and powerful combination to use.

A Linux container is a lightweight virtualization technique that provides isolation in the form of namespaces and resource control using cgroups. It's often described as chroot on steroids. The underlying host kernel should support these features for you to create containers. Unlike in the case of VMs, there is no hypervisor layer; instead, there is direct interaction with the hardware components. Finally, since every container is just a process, bootstrapping containers is very quick (approximately 1 second) and so is bringing them down.

Let's now look at how Ansible can be used with Docker to make this a powerful working combination. In this section, we'll look at how we can use Ansible to perform the following:

- Installing Docker on hosts
- Deploying new Docker images
- Building or provisioning new Docker images

Installing Docker on hosts

Let's say you have a new host on which you want to start creating Docker containers. In order to set it up, one of the first things you might want to do is actually install Docker. The Ansible website has an example of a Docker role for Ubuntu. We'll look at a relatively simple example of setting up Docker on CentOS 7. CentOS 7 supports Docker and is included in the CentOS-Extras repository. For those who are not familiar, CentOS-Extras is one of the repositories that come in by default when you install CentOS 7, hence, you can directly install it using yum install docker.

In CentOS 6, the Docker package was in the EPEL repository and, as a result, you needed to add the repository before pulling the repository. You can add the EPEL repository as follows:

```
$ rpm –Uvh http://download.fedoraproject.org/pub/epel/6/x86_64/epel-
release-6-8.noarch.rpm
```

The package name is docker-io in RHEL 6 / CentOS 6. You can install it with yum install docker-io.

So, we'll create an install_docker role with a single main.yml file. The contents of the file are shown in the following screenshot:

```
$ cat install_docker/tasks/main.yml
---
  - name: install docker package
    yum: name=docker state=present
    sudo: yes

  - name: enable docker service
    service: name=docker enabled=yes state=started
    sudo: yes
$ cat site.yml
---
- hosts: dockerhost
  user: vagrant
  sudo: yes
  roles:
    - install_docker
```

This is straightforward, isn't it? We'll now run `site.yml` and see what happens:

```
$ ansible-playbook -i inventory --private-key ~/.ssh/ansible_key site.yml

PLAY [dockerhost] ***********************************************************

GATHERING FACTS ************************************************************
ok: [192.168.33.16]

TASK: [install_docker | install docker package] ****************************
changed: [192.168.33.16]

TASK: [install_docker | enable docker service] *****************************
changed: [192.168.33.16]

PLAY RECAP *****************************************************************
192.168.33.16              : ok=3    changed=2    unreachable=0    failed=0
```

No hitches. Docker gets installed on the machine. We can check if it's installed by running a couple of verification commands on the machine. (Remember serverspec? Make sure you write serverspec tests!)

The following is what we observe when we validate the installation of Docker on the machine:

```
[root@vagrant-centos7 ~]# ps aux | grep docker
root      3586  0.1  0.7 424324  7900 ?        Ssl  08:13   0:00 /usr/bin/docker -d --selinux-enabled
root      3701  0.0  0.0 112640   964 pts/0    R+   08:14   0:00 grep --color=auto docker
[root@vagrant-centos7 ~]# docker
Usage: docker [OPTIONS] COMMAND [arg...]
 -H=[unix:///var/run/docker.sock]: tcp://host:port to bind/connect to or unix://path/to/socket to use

A self-sufficient runtime for linux containers.

Commands:
    attach    Attach to a running container
    build     Build a container from a Dockerfile
    commit    Create a new image from a container's changes
    cp        Copy files/folders from the containers filesystem to the host path
    diff      Inspect changes on a container's filesystem
    events    Get real time events from the server
    export    Stream the contents of a container as a tar archive
    history   Show the history of an image
    images    List images
    import    Create a new filesystem image from the contents of a tarball
    info      Display system-wide information
    inspect   Return low-level information on a container
    kill      Kill a running container
    load      Load an image from a tar archive
    login     Register or Login to the docker registry server
    logs      Fetch the logs of a container
    port      Lookup the public-facing port which is NAT-ed to PRIVATE_PORT
    ps        List containers
```

The Docker service is up-and-running and we can now deploy or create new containers on the machine. In the preceding case, we have a single Docker host, but the same playbook can be run across a multitude of hosts as well, provided you have all those hosts in the inventory file.

Deploying new Docker images

Docker has a public registry (`https://registry.hub.docker.com/`) that has thousands of images. Our next step is to check whether we can download those images and run them locally. We'll use Ansible to do this for us using the `docker` module. We create a role called `nginx` and add a `main.yml` file that does the job for us. We'll be pulling the Nginx image for this purpose. Let's look at the following code:

```
$ cat nginx/tasks/main.yml

---

  - name: Run nginx docker container
    docker: image=nginx command="nginx" hostname=nginx_container
        ports=80


  - name: Display IP address and port mappings for docker container
    debug: msg={{ inventory_hostname }}:{{
        item['HostConfig']['PortBindings']['80/tcp'][0]['HostPort']
        }}
    with_items: docker_containers
```

Here, the first task is the one that does all the magic:

- `image=nginx`: This indicates that we'll download the nginx image
- `command=nginx`: This tells us what command needs to be run to start the Docker container
- `hostname=nginx_container`: This tells us the name of the Docker container
- `ports=80`: This tells us that the container will run on port `80`

The `docker_containers` variable is populated in the first step once the container or containers are created and is an array of containers. The second task uses the `docker_containers` array to print the IP address and the host port that is mapped to port 80 of the container that is created. Remember that all containers are attached to a separate bridge and there is port forwarding from the host to the container, by default. We'll add the `nginx` role after the `install_docker` role in `site.yml` as follows:

```
$ cat site.yml
---
- hosts: dockerhost
  user: vagrant
  sudo: yes
  roles:
    - install_docker
    - nginx
```

Without further ado, let's run `site.yml`. When we run the playbook, it throws the following error:

```
TASK: [nginx | Run nginx docker container] **********************************
failed: [192.168.33.16] => {"failed": true, "item": ""}
msg: `docker-py` doesn't seem to be installed, but is required for the Ansible Docker module.

FATAL: all hosts have already failed -- aborting
```

This basically means that the `docker-py` package isn't installed on the Docker host and that it is a prerequisite for using the `docker` module. We go back and modify the `install_docker` role's `main.yml` file as follows:

```
$ cat install_docker/tasks/main.yml
---
  - name: install docker package
    yum: name=docker state=present

  - name: enable docker service
    service: name=docker enabled=yes state=started

  - name: install pip
    easy_install: name=pip

  - name: install docker-py package
    pip: name=docker-py
```

On running the entire `site.yml` playbook again, we will get to see what we want to see!

What we've shown in the preceding screenshot is the truncated output. Essentially, Docker outputs the entire metadata associated with the new Docker image. Let's look at the IP and port mappings that were part of the task:

Let's check whether the container is well and truly up:

```
[vagrant@vagrant-centos7 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                   NAMES
33b45a1d5157        nginx:1             nginx               9 seconds ago       Up 9 seconds        0.0.0.0:49153->80/tcp   agitated_morse3
```

Now, let's say you want to set up multiple Docker containers. This is typically the case when you want to set up multiple web servers. In this case, we modify `main.yml` in tasks by adding the parameter `count=3`. Everything else remains the same.

```
$ cat nginx/tasks/main.yml
---
  - name: Run nginx docker container
    docker: image=nginx command="nginx" hostname=nginx_container ports=80 count=3

  - name: Display IP address and port mappings for docker container
    debug: msg={{ inventory_hostname }}:{{ item['HostConfig']['PortBindings']['80/tcp'][0]['HostPort'] }}
    with_items: docker_containers
```

On running the Ansible playbook (no change in command), we end up with three Docker containers. The output is too big but on copying the output to a file (called `output`) and grepping for `msg`, we find that three containers have been created:

```
$ grep msg output
    "msg": "192.168.33.16:49154"
    "msg": "192.168.33.16:49155"
    "msg": "192.168.33.16:49153"
```

As you can see, the ports that are mapped on the host are different but the actual ports on the containers themselves would be 80. Let's check the host to see if there are three containers, each running on port 80:

```
[vagrant@vagrant-centos7 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                   NAMES
4b3c5dc76519        nginx:1             nginx               About a minute ago  Up About a minute   0.0.0.0:49155->80/tcp   evil_elion3

1715983a8af3        nginx:1             nginx               About a minute ago  Up About a minute   0.0.0.0:49154->80/tcp   dreamy_elion1

25b4df2e60b1        nginx:1             nginx               About a minute ago  Up About a minute   0.0.0.0:49153->80/tcp   elegant_fermi6
```

Yay! This worked as expected. There are other parameters as well that you can explore with respect to the `docker` module but we've already covered how to get started with the module.

Building or provisioning new Docker images

Let's now look at a more interesting use case: building the Docker image using Ansible. If you remember, in an earlier chapter we created a build agent role and installed Java, Maven, and Ant as part of the build agent. Now, assume that your Lead has read about Docker and wants to introduce Docker somewhere in the organization, and figures out that the first place this can be done is in the build system. The Lead figures out that adding new build agents that are Docker images would be a great way to test it. The ball is now in your court and you're supposed to come up with Docker build agents.

Before we get into that, we need to understand how a Docker container is built. In order to build a Docker container, you need to provide a Dockerfile. The Dockerfile will have one or more commands that will be run in a sequence to generate the Docker image. Every command that is run in the Dockerfile generates a new image and is cached. So if you have four steps in a Dockerfile, you will end up with four containers. Now, if you build a second container from the same base image with four steps, where the first two steps are held in common with the previous container, then it will reuse the cached containers of the first and won't rebuild fresh containers for the first two steps. At each step, the output container of the previous step is the input. We'll look at a Dockerfile in our next step.

So you start playing with Docker, figure out how a Dockerfile works, and want to create a Docker image for your build agent. Now, instead of having a set of RUN commands in the Dockerfile, you want to reuse what you've already done with Ansible and more importantly you want to make sure that the commands present in the Dockerfile follow a particular format. So, what we recommend is, whether you're using Ansible, Chef, or Puppet, make sure you run the necessary configuration management tool command in the Dockerfile.

Let's take a look at how our Dockerfile looks once we embed the Ansible playbook contents, as shown in the following screenshot:

```
[vagrant@vagrant-centos7 chapter3_role2]$ ls
Dockerfile  Vagrantfile  build_agent.yml  cassandra      importantfile  jdk           playbooks  spec
Rakefile    build_agent  build_agent_2    cassandra.yml  inventory      passwords.yml  site.yml
[vagrant@vagrant-centos7 chapter3_role2]$ cat Dockerfile
FROM centos:centos6
RUN rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
RUN yum install -y ansible
ADD . /tmp
WORKDIR /tmp
RUN ansible-playbook build_agent.yml -i inventory -c local
ENV PATH $PATH:/opt/apache-maven-3.0.5/bin:/opt/apache-ant-1.7.1/bin
ENV ANT_HOME /opt/apache-ant-1.7.1
```

As you can see in the previous screenshot, a Dockerfile consists of:

- `FROM centos:centos6`: This specifies the base image from which you'll build the Docker image. In this case, we chose a base CentOS 6 image.

- `RUN`: This basically runs the commands on the containers and generates new intermediate containers. In this case, it generates a new container after steps 2 and 3.

- `ADD`: This command allows you to copy the contents of the current directory to the container that is being built in the specified location. Here, it copies the current directory to `/tmp`.

- `WORKDIR`: This sets the current working directory in the container to `/tmp`.

- `RUN ansible-playbook`: This command runs the `ansible-playbook` command from `/tmp`.

- `ENV`: This sets the environment variables. In this case, it sets the `PATH` and `ANT_HOME` variables.

To summarize, we copy the contents of the current directory to `/tmp` and run the `ansible-playbook` command from there by providing the `build_agent.yml` file as a parameter. This `build_agent.yml` file was copied over when we ran the `ADD` command.

Let's now build the image. We run the command `docker build -t build_agent .`. This means that we will now build a Docker container from a Dockerfile that is present in the current directory, and the Docker image that we build will be tagged `build_agent`. We will now run the following command:

```
$ docker build -t build_agent .
Uploading context 135.7 kB
Uploading context
Step 0 : FROM centos:centos6
 ---> b1bd49907d55
Step 1 : RUN rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-
release-6-8.noarch.rpm
 ---> Running in fe3bea02fca0
warning: /var/tmp/rpm-tmp.faqlEl: Header V3 RSA/SHA256 Signature, key ID
0608b895: NOKEY
Retrieving http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-
release-6-8.noarch.rpm
Preparing...              ###################################################
epel-release             ###################################################
 ---> 74a7cccd4ebd
```

```
Removing intermediate container fe3bea02fca0

Step 2 : RUN yum install -y ansible

 ---> Running in 4efed0c7fe20

 --> Finished Dependency Resolution


Dependencies Resolved


========================================================================
=======
 Package               Arch         Version            Repository
Size
========================================================================
=======
Installing:
 ansible               noarch       1.7-1.el6          epel
874 k
Installing for dependencies:
 PyYAML                x86_64       3.10-3.el6         epel
157 k
 libyaml               x86_64       0.1.6-1.el6        epel
52 k
 python-babel          noarch       0.9.4-5.1.el6      base
1.4 M
 python-crypto         x86_64       2.0.1-22.el6       base
159 k
 python-crypto2.6      x86_64       2.6.1-1.el6        epel
530 k
 python-httplib2       noarch       0.7.7-1.el6        epel
70 k
 python-jinja2         x86_64       2.2.1-2.el6_5      updates
466 k
 python-keyczar        noarch       0.71c-1.el6        epel
219 k
 python-paramiko       noarch       1.7.5-2.1.el6      base
728 k
 python-pyasn1         noarch       0.0.12a-1.el6      base
70 k
 python-setuptools     noarch       0.6.10-3.el6       base
336 k
```

```
Transaction Summary
============================================================================
=======
Install      12 Package(s)
< Removing information around handling dependencies >
Installed:
  ansible.noarch 0:1.7-1.el6


Dependency Installed:
  PyYAML.x86_64 0:3.10-3.el6            libyaml.x86_64 0:0.1.6-1.el6
  python-babel.noarch 0:0.9.4-5.1.el6     python-crypto.x86_64 0:2.0.1-
22.el6
  python-crypto2.6.x86_64 0:2.6.1-1.el6   python-httplib2.noarch
0:0.7.7-1.el6
  python-jinja2.x86_64 0:2.2.1-2.el6_5    python-keyczar.noarch 0:0.71c-
1.el6
  python-paramiko.noarch 0:1.7.5-2.1.el6   python-pyasn1.noarch
0:0.0.12a-1.el6
  python-setuptools.noarch 0:0.6.10-3.el6


Complete!
 ---> 10daf9ce140f
Removing intermediate container 4efed0c7fe20
Step 3 : ADD . /tmp
 ---> 426776c7f5e4
Removing intermediate container 45681c651661
Step 4 : WORKDIR /tmp
 ---> Running in ccd1c5a489d8
 ---> 8384bee9428f
Removing intermediate container ccd1c5a489d8
Step 5 : RUN ansible-playbook build_agent.yml -i inventory -c local
 ---> Running in 643c96b9c9ba
 [WARNING]: The version of gmp you have installed has a known issue
regarding
timing vulnerabilities when used with pycrypto. If possible, you should
update
it (ie. yum update gmp).
```

```
PLAY [build_agents] **************************************************
******

GATHERING FACTS *****************************************************
******
ok: [localhost]


TASK: [jdk | download jdk rpm] **************************************
******
changed: [localhost]


TASK: [jdk | setup java jdk] ****************************************
******
changed: [localhost]


TASK: [build_agent | install which and tar packages]
*************************
changed: [localhost] => (item=which,tar)


TASK: [build_agent | create ant directory] ************************
******
ok: [localhost]


TASK: [build_agent | download ant] *******************************
******
changed: [localhost]


TASK: [build_agent | untar ant] **********************************
******
changed: [localhost]


TASK: [build_agent | add ant to /etc/profile.d]
******************************
changed: [localhost]


TASK: [build_agent | download maven] ******************************
******
changed: [localhost]
```

```
TASK: [build_agent | untar maven] *************************************
******

changed: [localhost]


TASK: [build_agent | add maven file to /etc/profile.d]
***********************

changed: [localhost]


PLAY RECAP ***********************************************************
******

localhost                    : ok=11    changed=9    unreachable=0
failed=0


 ---> 071108e3325e

Removing intermediate container 643c96b9c9ba

Step 6 : ENV PATH $PATH:/opt/apache-maven-3.0.5/bin:/opt/apache-
ant-1.7.1/bin

 ---> Running in 95132f21b8da

 ---> 00e21ce3d0da

Removing intermediate container 95132f21b8da

Step 7 : ENV ANT_HOME /opt/apache-ant-1.7.1

 ---> Running in 06b55dedac49

 ---> a311af7b2f84

Removing intermediate container 06b55dedac49

Successfully built a311af7b2f84
```

There you go! We have a Docker image that's been built and is tagged `build_agent`.
Let's check if the `build_agent` image exists:

```
$ docker images
REPOSITORY         TAG              IMAGE ID          CREATED
VIRTUAL SIZE

build_agent        latest           a311af7b2f84      32 minutes
ago       659.9 MB
centos             centos6          b1bd49907d55      4 weeks ago
212.5 MB
```

Let's create a container from this image and check if the image has Maven, Ant, and Java installed. Finally, check the environment variables that we set:

```
[vagrant@vagrant-centos7 chapter3_role2]$ docker run -i -t build_agent /bin/bash
bash-4.1# mvn
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.089s
[INFO] Finished at: Mon Sep 01 09:36:11 BST 2014
[INFO] Final Memory: 2M/15M
[INFO] ------------------------------------------------------------------------
[ERROR] No goals have been specified for this build. You must specify a valid lifecycle phase or a goal in the format <plugin-
ifact-id>[:<plugin-version>]:<goal>. Available lifecycle phases are: validate, initialize, generate-sources, process-sources,
 process-classes, generate-test-sources, process-test-sources, generate-test-resources, process-test-resources, test-compile,
ckage, pre-integration-test, integration-test, post-integration-test, verify, install, deploy, pre-site, site, post-site, site
 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/NoGoalSpecifiedException
bash-4.1# ant
Buildfile: build.xml does not exist!
Build failed
bash-4.1# java -version
java version "1.6.0_45"
Java(TM) SE Runtime Environment (build 1.6.0_45-b06)
Java HotSpot(TM) 64-Bit Server VM (build 20.45-b01, mixed mode)
bash-4.1# echo $ANT_HOME
/opt/apache-ant-1.7.1
bash-4.1# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/opt/apache-maven-3.0.5/bin:/opt/apache-ant-1.7.1/bin
```

Bingo! Everything works as expected. The next thing you want to do is create containers from the `build_agent` image and use them as part of your build system.

With this, we come to the end of the Docker section. There are many more features that you can explore on the Docker website. We wanted to highlight how you can use Docker to open predefined images and prepare new images for certain configurations using Ansible. Hope you got a good introduction into how you can do this in your environment.

Dynamic Inventory

You might be wondering as to why we're covering a topic such as Dynamic Inventory in a chapter on Provisioning. This is because, whenever you bring up a new machine, you need to account for that as part of your new modified inventory, and, the next time there is a requirement to find out what's been provisioned in the cloud or your data center, you need to make sure that the new machine or machines are absolutely accounted for. So, let's look at Dynamic Inventory in a little more detail.

An inventory is a source of truth for Ansible. Maintaining the inventory would not be difficult when you have a small set of servers running, that is around 10 to 20 servers. However, imagine having a vast environment with thousands of servers running; maintaining a huge list of servers manually would be a nightmare, might result in an inaccurate inventory, and would definitely lead to errors.

To deal with such a tedious job, Ansible allows you to use a dynamic script, through which you can generate an inventory on-the-fly. Ansible provides an inventory script for the majority of the cloud providers out there (`https://github.com/ansible/ansible/tree/devel/plugins/inventory`) but, if you still do not find one for your cloud provider or data center, you can write a script of your own that can use your cloud provider's APIs and generate an inventory out of that. The Ansible inventory scripts can be written in any language but, in most cases, Dynamic Inventory scripts are written in Python. These scripts should be executable in nature (`chmod + x inventory.py`).

In this section, we will take a look at Amazon's `ec2` inventory script and try to plug it in with Ansible. You need to download the following two files from Ansible's GitHub repository:

- The `ec2.py` inventory script
- The `ec2.ini` file, which contains the configuration for your `ec2` inventory script

Ansible uses **boto** to communicate with Amazon using APIs. To allow this communication, you need to export the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` variables.

You can use the inventory in two ways:

- Pass it directly to an `ansible-playbook` command using the `-i` option and copy the `ec2.ini` file to your current directory where you are running the Ansible command
- Copy the `ec2.py` file to `/etc/ansible/hosts`, make it executable using `chmod +x`, and copy the `ec2.ini` file to `/etc/ansible/ec2.ini`

The `ec2.py` file will create multiple groups based on the region, availability zone, tags, and so on. You can check the contents of the inventory file by running `./ec2.py --list`.

Let's see an example playbook with the `ec2` Dynamic Inventory, which will take the NodeName and restart the `hadoop-hdfs-datanode` service on it.

```
$ cat playbooks/hadoop_data_restart.yml
---
-   hosts: "{{ nodes }}"
    remote_user: ec2-user
    tasks:
        - name: Restart hadoop-hdfs-datanode service
          service: name=hadoop-hdfs-datanode state=restarted
          sudo: yes
```

The preceding playbook will take the node/group name and restart the `hadoop-hdfs-datanode` service on it. Let's see how the playbook runs:

```
$ ansible-playbook -i ec2.py playbooks/hadoop_data_restart.yml -e "nodes=tag_name_hadoop-data"

PLAY [tag_name_hadoop-data] *********************************************

GATHERING FACTS ********************************************************
ok: [ec2-54-227-43-141.compute-1.amazonaws.com]

TASK: [Restart hadoop-hdfs-datanode service] **************************
changed: [ec2-54-227-43-141.compute-1.amazonaws.com]

PLAY RECAP ************************************************************
ec2-54-227-43-141.compute-1.amazonaws.com : ok=2    changed=1    unreachable=0    failed=0
```

In the preceding example, we're using the `ec2.py` script instead of a static inventory file with the `-i` option and the group name `tag_name_hadoop-data`, which contains all the hosts tagged as `hadoop-data`. Right now, we just have one host with that tag, which we launched earlier during the `ec2` provision example.

Similarly, you can use these inventory scripts to perform various types of operations. For example, you can integrate it with your deployment script to figure out all the nodes in a single zone and deploy to them if you're performing your deployment zone-wise (a zone represents a data center) in AWS.

If you simply want to know what all the web servers in the cloud are and you've tagged them using a certain convention, you can do that by using the Dynamic Inventory script by filtering out the tags. Furthermore, if you have special scenarios that are not covered by the script that is already present, you can enhance the script to provide the required set of nodes in the JSON format and you can act on those set of nodes from the playbooks. If you're using a database to manage your inventory, your inventory script can query the database and dump a JSON. It could even sync with your cloud and update your database on a regular basis.

Summary

With this, we come to the end of this chapter on Provisioning. We covered how we can spawn machines in the cloud, particularly in Amazon AWS and DigitalOcean; how we can use Docker to create new images and provision containers; and, finally, how we can use a dynamic inventory to handle a cloud infrastructure in a robust manner.

Next, we'll proceed to an important chapter on Deployment and Orchestration. We'll look at how we can deploy applications and adhere to certain best practices in the Deployment world using Ansible. We strongly advise you to get yourself a coffee as the last chapter (excluding the *Appendix*, *Ansible on Windows, Ansible Galaxy, and Ansible Tower*) promises to be heavy! During your coffee break, here are a few questions that you can think about:

- How would you use Ansible cloud modules in your environment? If there are no Ansible modules for your cloud, by combining knowledge from the previous chapter and this chapter can you write out custom modules for your cloud?

- How can you use a dynamic inventory in your environment? Do you even have a need for a dynamic inventory?

- Can you use Docker in your environment if you already aren't doing it? If yes, think about how you can use Ansible to generate Docker images and run them.

7
Deployment and Orchestration

"Going through a deployment! BACK OFF!", says one of our colleagues who also plays the part of a Release Engineer. We've had responses such as this and others that can't be written from colleagues in closed rooms while performing deployments. During deployment events, we've had:

- People writing instructions on the board
- Tense colleagues sipping their tea or coffee and listening intently to these instructions
- People on a conference call for hours
- Celebration after every component is deployed
- Finally, a lot of people coming out as if they actually went through a "war' in the war room
- Hell breaks loose if there has to be a rollback

We can't promise that we'll change many of these, but we will include certain best practices that might reduce the pain of deployment. However, we'll mostly focus on the aspects of Ansible that can be used to ease your deployment and orchestration headache on the release day of software products.

Let's start by defining deployment and orchestration in this context as follows:

- **Deployment**: `Freedictionary.com` defines Deployment as follows:

  To position (troops) in readiness for combat, as along a front or line.

  Well, we guess that's why people get into a "war room" to perform software deployments. Software deployment is a way of releasing new code on machines to perform certain tasks to make sure the overall system works as intended. Having been in several deployment war rooms before, we can guarantee that completing deployment activities without any glitch is a source of immense satisfaction and happiness.

- **Orchestration**: Orchestration is defined as:

  Arrangement of music for performance by an orchestra.

  Deployment orchestration is similar to the preceding definition, because here we have to make sure that the right set of systems has the right versions and are updated in the right order, so that, in the end, they work together as a cohesive system to provide optimum performance. The art of making this happen is orchestration.

In this chapter, we'll cover the following topics:

- Deploying a sample Ruby on Rails application
- Packaging
- Deployment: points to consider
- Canary deployment
- Orchestration and deployment: deployment of Tomcat application

Deploying a sample Ruby on Rails application

We'll start with deploying a sample Ruby on Rails application. Ruby on Rails is a popular Ruby framework for web development and is widely used. A similar deployment pattern is followed for Node.js, PHP, and Python applications as well. Hence, the steps that we show here for Rails would be similar to other language deployments.

We'd like you to consider the philosophy or approach that we follow here for your applications. It might help to read briefly about how Ruby on Rails works before reading this section further, but we'd like to stress that the approach is the same for similar frameworks. The example mentioned here is based on Rails and, in the later part of the chapter, we will also include Java deployments. Typical steps that are followed in such applications are as follows:
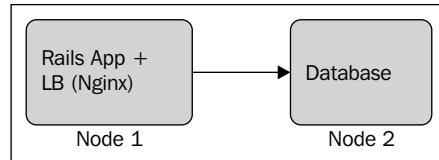
- **Setting up a database**: There are multiple ways to set up databases. Broadly, they can be classified as:
  - Single node, which acts as a master
  - Multinode, which acts either as master-slave (leader-follower is the term used by Django, a popular Python framework) or master-master

- **Setting up the application on one or more nodes**: The most common method of deployment is to directly download the files from Git and use either the latest commit from the `master` branch, or a particular tag from the `master` branch, and start the application server. This model can be followed for all those applications that don't need a compiled version to specifically deploy, such as a `war` or `jar` file.

  Heroku is a popular platform where you can host Rails, Node.js, or Python applications; they ask you to provide the link to your Git repository master, and the rest is taken care of by them. Coming back, in Rails, there are multiple options for the Rails server, right from what you can run in development environments such as **Thin** (`http://code.macournoyer.com/thin/`) and **Webrick** (`http://en.wikipedia.org/wiki/WEBrick`) to more popular production servers such as **Passenger** (`https://www.phusionpassenger.com/`), **Unicorn** (`https://github.com/blog/517-unicorn`), and **Puma** (`http://puma.io/`).

  Servers such as Unicorn also support zero downtime deployments when you're performing releases by deploying the new version and then sending a USR2 signal to the process to reload the new code. Each of these servers runs on multiple worker processes that execute your request.

- **Setting up the load balancer**: As you scale, you will have more number of application servers and as a result you need a frontend load balancer (reverse proxy), which will route your requests to application servers, get the responses, and send them back to your end users. We won't consider advanced functionality, such as SSL termination, in this chapter.

Without further ado, let's start with a single two-node deployment of a sample Rails application, whose diagrammatic representation is shown as follows:

We start at the point where we have two Amazon EC2 instances provisioned. You can refer to the previous topic on Provisioning to see how to set up servers on AWS. In this example, we will perform the deployment on these servers.

First things first. When we start deploying applications, as we mentioned in *Chapter 3*, *Taking Ansible to Production*, we would like to deal with all interactions on various servers using Roles. So, in this case, we will have two roles, described as follows:

- The `db` role: This role is responsible for setting up the database. Consider the following example:

```
$ cat db.yml
  ---
- hosts: db
  user: ec2-user
  sudo: yes
  roles:
     - db
```

- The `app_server` role: The application along with the load balancer will be set up by this role. The command line is as follows:

```
$ cat app_server.yml
  ---
- hosts: app
  user: ec2-user
  roles:
     - app_server
```

Typically, there is also a `common` role that we covered in *Chapter 3*, *Taking Ansible to Production*, on Roles. The `common` role will take care of security parameters, common packages, ntp settings, dns settings, and so on. We'll ignore these factors here as our focus is on the application. There are three task files for `db`. We will look at `main.yml` and one of the included files, `pre_requisites.yml`, to start with:

```
$ cat db/tasks/main.yml
---
- include: pre_requisites.yml
- include: mysql_server.yml
$ cat db/tasks/pre_requisites.yml
---
- name: Install pre-requisite packages
  yum: pkg={{ item }} state=present
  with_items:
     - MySQL-python
```

Let's now look at the third file, `mysql_server.yml`.

You might wonder where the `mysql55-server` came from. This is a standard package available in the Amazon repository and we used Amazon Linux for these deployment examples on EC2, as mentioned earlier.

Now, consider the following screenshot:

```
$ cat db/tasks/mysql_server.yml
---
- name: install mysql server
  yum: pkg={{ item }} state=present
  with_items:
     - mysql55-server

- name: start mysqld server
  service: name=mysqld state=started

- name: update mysql root password for all root accounts
  mysql_user: name=root host={{ item }} password={{ mysql_root_password }}
  with_items:
     - "{{ inventory_hostname }}"
     - 127.0.0.1
     - localhost

- name: copy .my.cnf file with root password credentials
  template: src=root/.my.cnf dest=/root/.my.cnf owner=root mode=0600

- name: ensure anonymous users are not in the database
  mysql_user: name='' host={{ item }} state=absent
  with_items:
     - localhost
     - "{{ inventory_hostname }}"

- name: remove the test database
  mysql_db: name=test state=absent

- name: set user privileges
  mysql_user: name=rails password={{ rails_user_password }} state=present priv='*.*:ALL'
```

In the preceding screenshot, the `.my.cnf` template file mentioned is present in `db/templates/root`.

We have two YAML files included in `main.yml`. They are as follows:

- `pre_requisites.yml`: In this file, we will include packages that need to be installed in order for the Ansible `myql_user` module to run them.
- `mysql_server.yml`: We used the `mysql_user` module, which is quite helpful. As you can see, the `mysql_root_password` function is parameterized. You can handle passwords either using Ansible Vault or via environment variables. In this case, we used environment variables, as follows.

```
$ cat db/vars/main.yml
---
mysql_root_password: "{{ lookup('env','MYSQL_ROOT_PASSWORD') }}"
rails_user_password: "{{ lookup('env','RAILS_USER_PASSWORD') }}"
```

> Lookup is another type of Ansible plugin that you can utilize. You can look up external data sources to obtain data, which can then be used either in Jinja templates or playbooks. In this case, we're looking up environment variables, but you can look up files and systems such as the `etcd` and `redis` keys; basically, you can also extend lookup plugins for your custom usage.

Finally, we're creating a Rails user that can be accessed from the application server. We've again used the `mysql_user` module to accomplish this. We'll now run `db.yml` to set up the database. This is demonstrated in the following screenshot:

```
$ ansible-playbook -i inventory db.yml --private-key key.pem

PLAY [db] ***********************************************************************

GATHERING FACTS ****************************************************************
ok: [54.91.13.212]

TASK: [db | Install pre-requisite packages] ************************************
ok: [54.91.13.212] => (item=MySQL-python)

TASK: [db | install mysql server] *********************************************
changed: [54.91.13.212] => (item=mysql55-server)

TASK: [db | start mysqld server] **********************************************
changed: [54.91.13.212]

TASK: [db | update mysql root password for all root accounts] *****************
ok: [54.91.13.212] => (item=54.91.13.212)
ok: [54.91.13.212] => (item=127.0.0.1)
ok: [54.91.13.212] => (item=localhost)

TASK: [db | copy .my.cnf file with root password credentials] *****************
ok: [54.91.13.212]

TASK: [db | ensure anonymous users are not in the database] *******************
ok: [54.91.13.212] => (item=localhost)
ok: [54.91.13.212] => (item=54.91.13.212)

TASK: [db | remove the test database] *****************************************
ok: [54.91.13.212]

TASK: [db | set user privileges] **********************************************
ok: [54.91.13.212]

PLAY RECAP *********************************************************************
54.91.13.212               : ok=9    changed=2    unreachable=0    failed=0
```

We now have the database set up. Let's check the grant permission for the `rails` user by logging into the mysql command line, as shown in the following screenshot:

```
mysql> use mysql
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql>
mysql> show  grants for 'rails'@'%' ;
+-----------------------------------------------------------------------------------------------------+
| Grants for rails@%                                                                                  |
+-----------------------------------------------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'rails'@'%' IDENTIFIED BY PASSWORD '*2470C0C06DEE42FD1618BB99005ADCA2EC9D1E19' |
+-----------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

It is now time to set up the application along with the load balancer! Let's look at our Ansible configuration in the following screenshot:

```
$ cat app_server/tasks/main.yml
---
- include: pre_requisites.yml
- include: app_install.yml
- include: nginx.yml
$ cat app_server/tasks/pre_requisites.yml
---
- name : Check if epel.repo exists
  stat: path=/etc/yum.repos.d/epel.repo
  register: repo

- name: Add epel yum repo
  shell: rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
  when: repo.stat.exists != true

- name: Install packages required
  yum: pkg={{ item }} state=present
  sudo: yes
  with_items:
     - git
     - ruby-devel
     - gcc
     - libxml2-devel
     - patch
     - sqlite-devel
     - mysql-devel
     - gcc-c++

- name: Install required system gems
  gem: name={{ item }} state=present user_install=no
  sudo: yes
  with_items:
     - bundler
```

We have three YAML files, one of which is shown in the preceding screenshot; the other two YAML files will be seen in the following screenshots. At a high level, `main.yml` calls three files: `pre_requisites.yml`, `app_install.yml`, and `nginx.yml`.

They are explained as follows:

- `pre_requisites.yml`: You can have a pattern that makes sure you always have `pre_requisites.yml`, which defines the preconfiguration. (We'll also look at Ansible's `pre_tasks` in the later part of this chapter.) In this case, the `pre_requisites.yml` file takes care of installing the development libraries that are required to install **gems** on the system. Gems, for those not familiar with Ruby, are custom library code that you can install and use as part of your application.

We will also install the `bundler` gem, system-wide, so that we can use it to install application-specific gems locally to the application directory. Tomorrow, if you share the same system for another Rails application in a parallel directory, you can run `bundle install` in that directory and it will install the gems related to that project in the respective directory. Similarly, in Node.js, you have `packages.json`, which defines the dependencies in the form of `npms` that needs to be installed and can be installed in a local (`node_modules`) directory within the main `nodejs` application folder. The `app_install.yml` playbook is shown in the following screenshot:

```
$ cat app_server/tasks/app_install.yml
- name: Checkout git repo
  git: repo=https://github.com/madhurranjan/ember-simple-auth-rails-demo.git
        dest={{ app_dir }}
  sudo: yes

- name: Change permissions of folder
  sudo: yes
  file: path={{ app_dir }} owner=ec2-user group=ec2-user recurse=yes state=directory

- name: Install gems via bundler
  command: bundle install chdir={{ app_dir }}

- name: Setup database.yml
  template: src=database.yml dest={{ app_dir }}/config/database.yml owner=ec2-user group=ec2-user mode=0644

- name: Set SECRET_KEY_BASE environment variable
  template: src=secret.sh dest=/etc/profile.d/secret.sh owner=root group=ec2-user mode=0644
  sudo: yes

- name: Run rake db:create
  shell: bundle exec /home/ec2-user/bin/rake db:create RAILS_ENV={{ rails_env }} chdir={{ app_dir }}

- name: Run rake db:migrate
  shell: bundle exec /home/ec2-user/bin/rake db:migrate RAILS_ENV={{ rails_env }} chdir={{ app_dir }}

- name: Run rake assets:precompile
  shell: bundle exec /home/ec2-user/bin/rake assets:precompile RAILS_ENV={{ rails_env }} RAILS_GROUPS=assets chdir={{ app_dir }}

- name: Setup unicorn.rb
  template: src=unicorn.rb dest={{ app_dir }}/config/unicorn.rb owner=ec2-user group=ec2-user mode=0644

- name: Start service
  shell: bundle exec /home/ec2-user/bin/unicorn_rails -p {{ app_port }} -E {{ rails_env }} -c {{ app_dir }}/config/unicorn.rb -d -D chdir={{ app_dir }}
```

- `app_install.yml`: This is the main YAML file that does the following tasks:
    - Downloading the code from the git repository (the `Checkout git repo` and `Change permissions of folder` tasks)
    - Installing folder-specific gems using bundler (the `Install gems via bundler` task)
    - Running Rails-specific commands, `rake db:create` and `rake db:migrate`, to set up the database (tasks: `Setup database.yml`, `Set SECRET_KEY_BASE environment variable`, `Run rake db:create`, and `Run rake db:migrate`)

In order to perform this activity, we transfer the `database.yml` template, which has the following content:

```
$ cat app_server/templates/database.yml
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test: &test
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000

production:
  adapter: mysql2
  database: production
  host: {{ groups['db'][0] }}
  port: 3306
  username: rails
  password: {{ rails_user_password }}
  pool: 5
  timeout: 5000

cucumber:
  <<: *test
```

The `host` parameter in the template file points to the database host that is present in the inventory file using the `groups['db'][0]` variable. Ansible allows you to refer to inventory groups in templates using `groups[name_of_inventory_group]`. If you'd like to access all the hosts, you can use `groups['all']`. This information is exposed as an array that can be used. Perform the following steps:

1. Precompile the assets (the `Run rake assets:precompile` task).

2. Set up the rails server (in this case, the Unicorn server) and start it (the `Setup unicorn.rb` and `Start service` tasks).

Next, we'll look at the load balancer configuration file, `nginx.yml`, which is given as follows:

```
$ cat app_server/tasks/nginx.yml
---
- name : Check if epel.repo exists
  stat: path=/etc/yum.repos.d/epel.repo
  register: repo
  tags: lb

- name: Add epel yum repo
  shell: rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
  when: repo.stat.exists != true
  tags: lb

- name: Add content to /etc/hosts
  lineinfile: dest=/etc/hosts state=present line="{{ ansible_eth0.ipv4.address }} {{ server_name }}"
  sudo: yes
  tags: lb

- name: Install nginx package
  yum: name=nginx state=latest
  sudo: yes
  tags: lb

- name: Start nginx server
  service: name=nginx state=started
  sudo: yes
  tags: lb

- name: Copy nginx conf
  template: src=sample_app.conf dest=/etc/nginx/conf.d/nginx.conf owner=nginx group=nginx mode=0644
  sudo: yes
  notify: restart nginx
  tags: lb
```

The preceding file downloads `nginx` and sets up the configuration file to point to the Unicorn socket (since it is set up on the same machine). The `nginx` configuration file, `nginx.conf`, which is set up using the `template` module, is shown in the following screenshot:

```
$ cat app_server/templates/sample_app.conf
upstream unicorn_server {
        server unix:{{ app_dir }}/tmp/sockets/unicorn.sock
        fail_timeout=0;
    }

    server {
        listen      80;
        server_name  {{ server_name }};
        root         {{ app_dir }}/public;

        #charset koi8-r;

        #access_log  /var/log/nginx/host.access.log  main;

        location / {
                try_files $uri @app;
        }

        location @app {
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_set_header Host $http_host;
                proxy_redirect off;
                # pass to the upstream unicorn server mentioned above
                proxy_pass http://unicorn_server;
        }
    }
}
```

Let's look at the variables that are configured as follows:

```
$ cat app_server/vars/main.yml

---

rails_user_password: "{{ lookup('env','RAILS_USER_PASSWORD') }}"

secret_base: "{{ lookup('env','SECRET_KEY_BASE') }}"

rails_env: production

app_dir: /opt/sample_rails_app

app_port: 3000

server_name: sample_app
```

We've already seen how `lookup` works earlier in the chapter. The sample application is set up in `/opt/sample_app`.

Now, let's run the `app_server.yml` role, as follows, to see whether we can achieve the intended result:

```
$ ansible-playbook -i inventory app_server.yml --private-key ~/.ssh/key.
pem


PLAY [app] *************************************************************
******


GATHERING FACTS *******************************************************
******
The authenticity of host '54.227.241.154 (54.227.241.154)' can't be
established.
RSA key fingerprint is 18:8b:ac:11:61:ed:cd:fc:f4:80:28:58:83:cd:f1:08.
Are you sure you want to continue connecting (yes/no)? yes
ok: [54.227.241.154]


TASK: [app_server | Check if epel.repo exists] ************************
******
ok: [54.227.241.154]


TASK: [app_server | Add epel yum repo] ********************************
******
skipping: [54.227.241.154]


TASK: [app_server | Install packages required] ************************
******
changed: [54.227.241.154] => (item=git,ruby-devel,gcc,libxml2-
devel,patch,sqlite-devel,mysql-devel,gcc-c++)


TASK: [app_server | Install required system gems]
****************************
changed: [54.227.241.154] => (item=bundler)


TASK: [app_server | Checkout git repo] ********************************
******
changed: [54.227.241.154]


TASK: [app_server | Change permissions of folder]
****************************
```

```
changed: [54.227.241.154]


TASK: [app_server | Install gems via bundler] **************************
******
changed: [54.227.241.154]


TASK: [app_server | Setup database.yml] ********************************
******
changed: [54.227.241.154]


TASK: [app_server | Set SECRET_KEY_BASE environment variable]
*****************
changed: [54.227.241.154]


TASK: [app_server | Run rake db:create] ********************************
******
changed: [54.227.241.154]


TASK: [app_server | Run rake db:migrate] *******************************
******
changed: [54.227.241.154]


TASK: [app_server | Run rake assets:precompile]
******************************
changed: [54.227.241.154]


TASK: [app_server | Setup unicorn.rb] **********************************
******
changed: [54.227.241.154]


TASK: [app_server | Start service] *************************************
******
changed: [54.227.241.154]


TASK: [app_server | Check if epel.repo exists] *************************
******
ok: [54.227.241.154]
```

```
TASK: [app_server | Add epel yum repo] ********************************
******

skipping: [54.227.241.154]


TASK: [app_server | Add content to /etc/hosts] ***********************
******

changed: [54.227.241.154]


TASK: [app_server | Install nginx package] ***************************
******

changed: [54.227.241.154]


TASK: [app_server | Start nginx server] ******************************
******

changed: [54.227.241.154]


TASK: [app_server | Copy nginx conf] *********************************
******

changed: [54.227.241.154]


NOTIFIED: [app_server | restart nginx] *******************************
******

changed: [54.227.241.154]


PLAY RECAP ***********************************************************
******

54.227.241.154              : ok=20    changed=17    unreachable=0
failed=0
```

Now, let's enter the URL and see if the application comes up, as shown in the
following screenshot:
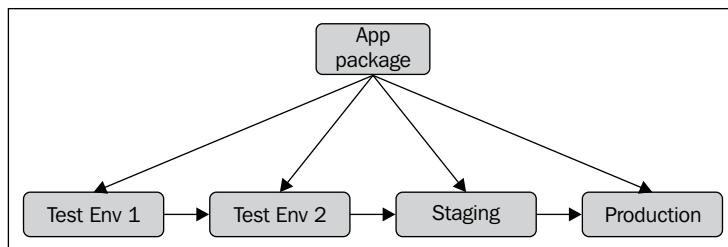
Yep, it does!

Packaging

Everything works, but is this the optimum way to deploy an application? If you show your sysadmin the scripts and packages we installed on our production machines, he/she will be deeply unhappy and will immediately raise a red flag. Ideally, you do not want to run any compiler and development packages on production machines and we have quite a few of them in `pre_requisites.yml` in our `app_server` role. It would then mean that it's acceptable to have all these libraries installed right up to the staging environment.

The sysadmin would be happy with this resolution but not the deployment team. That's because one of the best practices is to make sure that you try and deploy using the same scripts and packages into each of your environments, right from your first test environment to staging and, finally, to production, as shown in the following screenshot:

> Installing compilers such as `gcc` and development libraries to get your app up-and-running depends on your software stack. You won't have this problem if it's Java since you're probably just deploying a WAR or JAR file. However, with frameworks such as Rails or Node.js, you will face this problem.
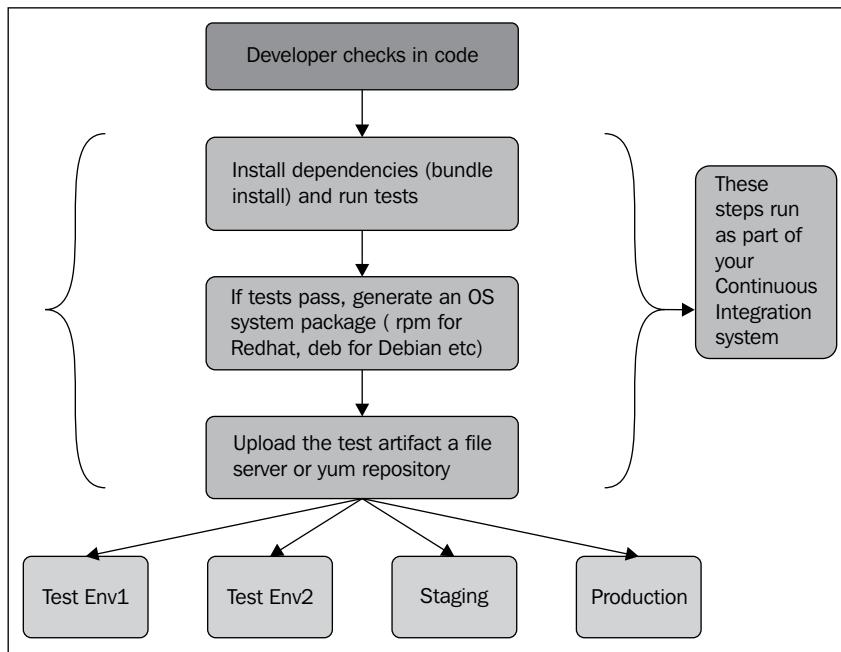
So, what do we do in such scenarios? Packaging is the answer. Every operating system has an optimized package manager that takes care of how a particular package should behave when it's installed. Let's take an example. In Red Hat systems, the **RPM Package Manager** (**RPM**) is used as a standard way to install packages. To generate RPMs, you need to provide an RPM spec file that defines the following parameters:

- What files and folders need to be in the RPM
- The layout of these files
- The precommands that need to be run before laying out these files

- The postcommands that need to be run after laying out the files on the system

- Which user should own the files

- What permissions should be associated with various files

Utilizing the goodness of package managers simplifies your entire deployment. Let's look at what happens when we apply the best practices of packaging to applications that are written in frameworks such as Rails and Node.js. The following diagram explains the flow you should adopt in order to use the best practices of the packaging world:

Developer checks in code

Install dependencies (bundle install) and run tests

If tests pass, generate an OS system package (rpm for Redhat, deb for Debian etc)

Upload the test artifact a file server or yum repository

These steps run as part of your Continuous Integration system

Test Env1

Test Env2

Staging

Production

Let's explain a couple of things in more detail. **yum** stands for the **Yellowdog Updater, Modified**. It is an open-source command-line package management utility for Linux operating systems using the RPM. Many enterprises create mirrors of public repositories within their organizations so that they don't pull dependencies from outside every single time. Similarly, you can create a custom repository to store your artifacts and install them as a system package.

> We won't be covering how to set up a custom `yum` repository server as it is out of the scope of this book, but the way you would consume it on the app machines would be as simple as adding a file in `/etc/yum.repos.d/<name of repo>.repo` followed by running the `yum makecache` and `yum install <name of package>` commands. An example of a repo file is as follows:
>
> ```
> $ cat /etc/yum/repos.d/custom.repo
> [Test]
> name=TestRepo
> baseurl=http://user:password@local.yumrepo.net/custom/
> centos6/x86_64/
> gpgcheck=0
> enabled=1
> ```

To generate `rpm`, you could use the traditional `spec` file or another popular open source tool called `fpm`. You can install `fpm` using `gem install fpm`, since it's a Ruby gem, after making sure that the `ruby-dev(el)` and `gcc` packages are installed on your operating system. There are other alternatives to `fpm` that help you create a package based on your package manager. We'll use `fpm` to generate the package and install it in our environment. You'll also see that the Ansible playbook to install the Rails app will be simplified to a much greater extent.

Let's now generate `rpm` from `fpm` and add it to our custom `yum` repository. You could also use other file servers or cloud storage services, such as S3, if you're not in a position to set up the custom package repositories. So let's now create the package of our application. Go to the application directory and run `bundle install`. (We've already covered how you can install a bundler in a system-wide manner earlier in the chapter.) Remember, we're still working with the Rails application that we started with. Let's install the required dependencies as follows:

```
$ cd ember-simple-auth-rails-demo
```

```
$ bundle install —local
```

The preceding command installs all the dependencies locally in the application directory. Next, we generate `rpm`. The `BUILD_NUMBER` value is a unique value that is fed from your CI system. For example, if you're using Jenkins, the `BUILD_NUMBER` value is automatically set for every run of the CI system. Consider the following command line, for example:

```
$ fpm -s dir -t rpm -v $BUILD_NUMBER -n sample_app -a all --exclude "**/
log/**"  --exclude "test/" --exclude "**/.git" -p /home/ec2-user -C
sample_rails_app --prefix /opt/sample_app_$BUILD_NUMBER .
```

We then copy `rpm` to the `yum` server and update the RPM database. The `createrepo` program creates a `repomd` (an XML-based RPM metadata) repository from a set of rpms. This command makes sure that the rpm would now be available on each of your servers when you run `yum install`. Consider the following command lines:

```
$ scp -i key.pem sample_app-${BUILD_NUMBER}-1.rpm <yum server>:/var/www/
html/centos6/x86_64  (
```

```
'/var/www/html/centos6/x86_64' is the path on the yum server where the
newly generated rpm has to be added.
```

```
$ ssh -i key.pem <yum server> "cd  /var/www/html/centos6/x86_64/;
createrepo --database /var/www/html/centos6/x86_64/"
```

Let's now see how our deployment changes based on the rpm approach. We'll first look at the modified Ansible playbooks as follows.

```
$ cat app_server/tasks/main.yml
---
- include: pre_requisites_2.yml
- include: app_install_2.yml
- include: nginx.yml
$ cat app_server/tasks/pre_requisites_2.yml
---
- name: Install mysql libs package
  yum: name=mysql55-libs state=latest
  sudo: yes

- name: Install required system gems
  gem: name={{ item }} state=present user_install=no
  sudo: yes
  with_items:
    - bundler
    - rake

- name : Check if internal.repo exists
  stat: path=/etc/yum.repos.d/internal.repo
  register: repo

- name: Add internal yum repo
  copy: src=internal.repo dest=/etc/yum.repos.d/internal.repo owner=root group=root mode=0644
  when: repo.stat.exists != true
  sudo: yes

- name: Run makecache
  shell: yum makecache
  when: repo.stat.exists != true
  sudo: yes
```

As you can see in the preceding screenshot, `prerequisites.yml` now has a couple of new tasks that take care of adding our internal repository to the machine where we're going to install the package. Let's look at `app_install_2.yml`. We changed `main.yml` in the preceding playbook to now include `app_install_2.yml` instead of `app_install.yml`. The modification is reflected in the following screenshot:

```
$ cat app_server/tasks/app_install_2.yml
---
- name: Install rails rpm
  yum: pkg=sample_app state=latest
  sudo: yes

- name: Change permissions of folder
  sudo: yes
  file: path={{ app_dir }}_{{ app_version }} owner=ec2-user group=ec2-user recurse=yes state=directory

- name: Setup database.yml
  template: src=database.yml dest={{ app_dir }}_{{ app_version }}/config/database.yml owner=ec2-user group=ec2-user mode=0644

- name: Set SECRET_KEY_BASE environment variable
  template: src=secret.sh dest=/etc/profile.d/secret.sh owner=root group=ec2-user mode=0644
  sudo: yes

- name: Run rake db:create
  shell: bundle exec /usr/local/bin/rake db:create RAILS_ENV={{ rails_env }} chdir={{ app_dir }}_{{ app_version }}

- name: Run rake db:migrate
  shell: bundle exec /usr/local/bin/rake db:migrate RAILS_ENV={{ rails_env }} chdir={{ app_dir }}_{{ app_version }}

- name: Setup unicorn.rb
  template: src=unicorn.rb dest={{ app_dir }}_{{ app_version }}/config/unicorn.rb owner=ec2-user group=ec2-user mode=0644

- name: Start service
  shell: bundle exec unicorn_rails -p {{ app_port }} -E {{ rails_env }} -c {{ app_dir }}_{{ app_version }}/config/unicorn.rb -d -D
 chdir={{ app_dir }}_{{ app_version }}
```

As you can see in the preceding screenshot, we've now changed our installation to directly install the dependencies from our internal `yum` repository. The application rpm is called `sample_app` and we've as usual used the `package` module to install it. We also don't have any development libraries installed. We require `mysql-libs` for the MySQL Gem but apart from that there is very little change that we have had to make. Let's look at the ansible run and see if the application comes up, using the following command lines.

```
$ ansible-playbook -i inventory --private-key ~/.ssh/key.pem app_server.
yml -e 'app_version=1'


PLAY [app] ********************************************************
******


GATHERING FACTS **************************************************
******
The authenticity of host '54.91.91.3 (54.91.91.3)' can't be established.
RSA key fingerprint is 59:89:d7:6a:f4:87:96:bc:37:3d:ee:1c:91:e3:cd:0b.
Are you sure you want to continue connecting (yes/no)? yes
```

```
ok: [54.91.91.3]


TASK: [app_server | Install mysql libs package]
*****************************
changed: [54.91.91.3]


TASK: [app_server | Install required system gems]
****************************
changed: [54.91.91.3] => (item=bundler)
changed: [54.91.91.3] => (item=rake)


TASK: [app_server | Check if internal.repo exists]
**************************
ok: [54.91.91.3]


TASK: [app_server | Add internal yum repo] ****************************
******
changed: [54.91.91.3]


TASK: [app_server | Run makecache] **********************************
******
changed: [54.91.91.3]


TASK: [app_server | Install rails rpm] ********************************
******
changed: [54.91.91.3]


TASK: [app_server | Change permissions of folder]
****************************
changed: [54.91.91.3]


TASK: [app_server | Setup database.yml] *******************************
******
changed: [54.91.91.3]


TASK: [app_server | Set SECRET_KEY_BASE environment variable]
****************
changed: [54.91.91.3]
```

```
TASK: [app_server | Run rake db:create] *******************************
******
changed: [54.91.91.3]


TASK: [app_server | Run rake db:migrate] ******************************
******
changed: [54.91.91.3]


TASK: [app_server | Setup unicorn.rb] *********************************
******
changed: [54.91.91.3]


TASK: [app_server | Start service] ************************************
******
changed: [54.91.91.3]


TASK: [app_server | Check if epel.repo exists] ************************
******
ok: [54.91.91.3]


TASK: [app_server | Add epel yum repo] ********************************
******
skipping: [54.91.91.3]


TASK: [app_server | Add content to /etc/hosts] ************************
******
changed: [54.91.91.3]


TASK: [app_server | Install nginx package] ****************************
******
changed: [54.91.91.3]


TASK: [app_server | Start nginx server] *******************************
******
changed: [54.91.91.3]


TASK: [app_server | Copy nginx conf] **********************************
******
```

```
changed: [54.91.91.3]


NOTIFIED: [app_server | restart nginx] ********************************
******

changed: [54.91.91.3]


PLAY RECAP ************************************************************
******

54.91.91.3                    : ok=20   changed=17   unreachable=0
failed=0
```

Let's check if the application starts and displays the following screenshot:

Indeed it does! The site comes up and we have a much better deployment method than before. You might have noticed that we pass app_version as a parameter to specify the version that needs to be installed. This corresponds to the build number that we used while generating the rpm. In the preceding case, every change results in a new package with a new version, and hence, a new deployment.

Is this the best way to handle new deployments? So far, we've come to the conclusion that generating standard system packages and deploying them is a better way of performing new deployments rather than deploying them based on source code, where we have to install several development libraries that system administrators might object to.

An additional note here is that we used shell commands to start processes. However, a better method is to encapsulate the process in the /etc/init.d scripts so that you handle the application lifecycle using the following commands:

- service app_name start: This is used to start the application.

- service app_name stop: This is used to stop the application.

- service app_name status: This is used to view the status of the application.

- service app_name restart: This will bring down the application and restart it. Typically, the way to do it is if you push code changes.

- `service app_name reload`: Certain underlying servers, such as Unicorn that we used in the preceding example, update the code and automatically reload the application if you send the process that is running a USR2 or HUP signal. Different servers behave differently and they might reload the applications differently. Some systems have the `hot deploy` option, which also allows you to perform seamless deployments. In such scenarios, running an `app_name` reload would be the ideal way to go about your application.

We've again borrowed the above idea from the way system packages behave. The standard for most system packages is to install them and then run a standard `service <name of package> start` command; it starts up the service. Similarly, run a command in order to stop the service. Your system administrators would be very happy to interact with their applications with commands such as these.

> We'll leave the `init` script as something for you to try out for the application that you're writing, but we hope that the logic of having `init` scripts is clear. Once you have the scripts ready, you can use a template or package the `init` script as well as part of the rpm and add a `service` task in your Ansible script to interact with your application.

Let's look at how to deploy new changes in more detail with system packages.

Deployment strategies with RPM

So far, in this chapter, we discussed how you can deploy an application for the first time on new machines. However, in most cases, we already have a certain version of the application that has been deployed and now, either to introduce a new feature or fix bugs, a new version has to be deployed. We'd like to discuss this scenario in greater detail.

At a high level, whenever we deploy an application, there are three kinds of changes to take care of:

- Code changes
- Config changes
- Database changes

The first two types are ideally handled by the RPM, unless you have very specific values that need to be set up during runtime. Files with passwords can be checked but they should be encrypted with Ansible Vault or dropped into files as templates during run time, just as we did with `database.yml`.

With templates, if the configs are ideally just name-value pairs that can be handled in a Jinja template, you should be good, but if you have other lines in the configuration that do not change, then it's better that those configuration files are checked and appear as part of the RPM. Many teams we've worked with check environment-specific folders that have all the configuration parameters; while starting the application, we provide the environment in which the application should be started.

Another way is to deploy the RPM with default values for all configuration properties while writing a different folder with name-value pairs that override the parameters in the default configuration that is part of the RPM.

> The database changes should be handled carefully. Ideally, you want them to be idempotent for a particular version of the RPM so that, even if someone tries to push database changes multiple times, the database is not really affected.
>
> For example, in the preceding case, we run `rake db:migrate` that is idempotent in nature; even if you run the same command from multiple machines, you shouldn't really face issues. The Rails framework does it by storing the database migration version in a separate table.

Having looked at the three types of changes, we can now examine how to perform `rpm` deployments for each release. When you're pushing new changes, the current version or service is already running. It's recommended that you take the server out of service before performing upgrades. For example, if it's part of a load balancer, make sure it's out of the load balancer and not serving any traffic before performing the upgrades. Primarily, there are the following two ways:

- Deploying newer versions of rpm in the same directory
- Deploying the rpm into a version-specific directory

Deploying newer versions of RPM in the same directory

There are two methods to deploy newer versions of rpm in the same directory. The first method flows in the following manner:

1. Stop the service.
2. Deploy the code.
3. Run database migrations.
4. Start the service.

The second method is as follows:

1. Deploy the code.
2. Run database migrations.
3. Perform a reload or restart operation to load the new code.

It's critical to make sure that the database changes don't break the current version or the new version that will be deployed. Remember that, while you're doing the database upgrades, at a particular point in time you will have certain servers serving old code and certain servers serving new code; you don't want to cause outages in either case. Let's also look at another approach.

Deploying the RPM in a version-specific directory

We use the best of what an RPM provides and what another Ruby deployment tool, **Capistrano**, does. Deploy the RPM into a version-specific directory, and the symlink into a common application folder. The advantage of having multiple RPMs installed and running out of a symlink is that every version will have its own directory, and the rollback, if there is one, is much simpler; you just have to change the symlink to point to the older version and restart the code. The remaining steps are similar. In Capistrano, you can specify the number of versions to maintain on the system at a time. With system packages, by default, you cannot have multiple versions installed but you can tweak this aspect to make sure multiple versions do get installed on your system. Yet another important aspect of this sort of deployment is that you can perform your deployment in the following phases:

- Push the new code to all servers prior to your deployment window.
  An extreme case can be deploying the RPM the day before the actual release.

- During the deployment downtime, perform the db upgrade, change the symlink, and restart the application.

These are ideas that you can definitely implement using Ansible but, though we presented the code to perform deployments to individual servers so far, we thought we should lay down the logic behind why someone would choose one method over the other. Let's now look at what Canary and Rolling deployments are.

Canary deployment

The name **Canary** is used with reference to the canaries that were often sent in to coal mines to alert miners about toxic gases reaching dangerous levels. Canary deployment is a technique that is used to reduce the risk of releasing a new version of software by first releasing it to a small subset of users (demographically, location-wise, and so on), gauging their reaction, and then slowly releasing it to the rest of the users.

Whenever possible, keep the first set of users as internal users, since it reduces the impact on the actual customers. This is especially useful when you introduce a new feature that you want to test with a small subset of users before releasing it to the rest of your customer base. If you're running, let's say, an application across multiple data centers and you're sure that certain users would only contact specific data centers when they access your site, you could definitely run a Canary deployment.

Orchestration of a Tomcat deployment

Now that we've seen how you can package code and deploy (and a couple of other possible ways to deploy) the package, we'll move on to multiple servers and show how you can orchestrate the deployment of a new version of code to your servers. This time, for the sake of variety, we'll show how you can perform a Tomcat deployment.

In this case, we will run an application, yet again on Amazon EC2 servers, but they could be running in your data center or on any other cloud. So far, we've spoken about how code should look on the box in terms of RPM and how to view database upgrades. Here, we'll show how you can remove servers out of service, upgrade them, and add them back to the mix, and we'll do it for all the servers in the mix, though in this case there are only four of them. This is popularly termed **Rolling Deployment**. If you only deploy new versions to a certain number of these servers based on either location or some other logic, you could view that as a Canary deployment.

Ansible allows you to perform rolling deployments. Let's look at the `update_tomcat.yml` playbook, as shown in the following screenshot:

```
$ cat playbooks/update_tomcat.yml
---
- name: Provisioning tomcat server
  hosts: "all"
  gather_facts: no
  user: "ec2-user"
  serial: 1
  sudo: yes

  pre_tasks:
    - name: Stopping monit
      service: name=monit state=stopped

    - name: Adding iptables rule to block port 8080
      shell: iptables -A INPUT ! -s 127.0.0.1 -p tcp -m tcp --dport 8080 -j DROP

    - name: Saving iptables rule
      shell: iptables-save

  roles:
    - { role: tomcat, tomcat_url: "http://www.us.apache.org/dist/tomcat/tomcat-7/v7.0.55/bin/apache-tomcat-7.0.55.tar.gz", tomcat_version: "7.0.55", tomcat_app: "https://tomcat.apache.org/tomcat-7.0-doc/appdev/sample/sample.war" }

  post_tasks:
    - name: Starting monit
      service: name=monit state=started

    - name: Flushing iptables rule
      shell: iptables -F
```

In the preceding playbook, we use roles to update the Tomcat server and application. In addition, we also use the `pre_task` and `post_task` features, which will make sure we don't disturb the production traffic by performing all prechecks and postchecks. These steps are part of the overall orchestration that Ansible allows you to perform. Most other tools do not provide these notions of pre- and post-tasks. Another feature that we've already seen is the `local_action` feature. Using these features, you can orchestrate your deployments in a much more controlled and elegant manner.

Getting back, the `pre_task` and `post_task` features can also be used with Roles while you're running regular configuration management tasks. The `pre_task` feature can send out notifications to the team responsible for monitoring, for example, by letting them know that an Ansible run would be starting on the machine and once the run is complete, as part of `post_task`, notify them about it. The most common use of the `pre_task` and `post_task` features is while performing releases. We'll look at these in more detail.

> Refer to the *Working with Roles* section of *Chapter 3*, *Taking Ansible to Production*, to know more about Ansible roles.

Let's look at the preceding playbook in greater detail.

As we will be rolling a deployment, we've specifically asked Ansible to process one host at a time by using the `serial` option within the playbook. Also, we will pass all hosts to Ansible because we only have four Tomcat servers. If you have a huge list of servers for your deployment, then you might want to create groups in your inventory (in such a way that it doesn't disturb your production traffic) and pass that group name to Ansible. In that case, you might also want to remove `serial: 1` to deploy to all servers in that specific group at the same time. For example, if your application is spread across multiple data centers and you're free to deploy an entire data center at one go, then you could create inventory groups based on the data center and release them to an entire data center at the same time. The following is the start of the playbook that contains the `serial` option that we spoke about:

```
- name: Provisioning tomcat server
  hosts: "all"
  gather_facts: no
  user: "ec2-user"
  serial: 1
  sudo: yes
```

Next, we will use `pre_tasks` to stop the `monit` service and take the Tomcat server out of the load balancer by applying the `iptables` rule. As the name suggests, the tasks listed under `pre_tasks` will be executed before the roles are executed. Apart from this, you can also add tasks that will disable monitoring checks, such as Nagios or Riemann, and send a maintenance notification to the right set of people. The `pre_tasks` task is given as follows:

```
  pre_tasks:
    - name: Stopping monit
      service: name=monit state=stopped

    - name: Adding iptables rule to block port 8080
      shell: iptables -A INPUT ! -s 127.0.0.1 -p tcp -m tcp --dport 8080
-j DROP

    - name: Saving iptables rule
      shell: iptables-save
```

> Note that we will block TCP port `8080` using `iptables` because the load balancer tries to perform a health check on this port. This trick is quite popular especially on AWS since it takes time to get the machine out of the load balancer (if you're using Amazon ELB for load balancing) and add it back in; by disabling and enabling a server, however, we are saving quite a bit of time.

We also use `post_tasks`, which will be executed after roles complete their execution. In this set of tasks, we restart `monit` and flush the `iptables` rule so that the load balancer can add the server back to its active pool. The `post_tasks` task is given as follows:

```
post_tasks:
  - name: Starting monit
    service: name=monit state=started


  - name: Flushing iptables rule
    shell: iptables -F
```

We use the `tomcat` role to update the Tomcat server and sample application. We also pass three variables: `tomcat_url`, `tomcat_version`, and `tomcat_app` to the role playbook that will be used to download the correct version of the Tomcat server and application. This is the first time we're calling the role in this way by explicitly passing the variable values along with the invocation of a role. Please look at the syntax carefully to make sure you understand it. It's also a way to generalize the role itself and pass the required parameters from outside. By doing this, whenever you want to change the version of Tomcat or the application, you just have to change one or more of these variables, but the actual role code doesn't really change. Further, each of the parameters, `tomcat_url`, `tomcat_version`, and `tomcat_app`, can themselves be variables that you could invoke here. This is left as an exercise for you. The `roles` task is given as follows:

```
roles:
  - { role: tomcat, tomcat_url: "http://www.us.apache.org/dist/tomcat/
tomcat-7/v7.0.55/bin/apache-tomcat-7.0.55.tar.gz", tomcat_version:
"7.0.55", tomcat_app: "https://tomcat.apache.org/tomcat-7.0-doc/appdev/
sample/sample.war" }
```

Let's now look at the `tomcat` role.

```
$ cat playbooks/roles/tomcat/tasks/main.yml
- name: Remove older version of tomcat
  file: path=/usr/local/tomcat state=absent

- name: Download Tomcat
  get_url: url={{ tomcat_url }} dest=/tmp

- name: Extract archive
  command: chdir=/tmp /bin/tar xvf apache-tomcat-{{ tomcat_version }}.tar.gz

- name: Copy tomcat folder
  command: creates=/usr/local/tomcat mv /tmp/apache-tomcat-{{ tomcat_version }}/ /usr/local/tomcat
  sudo: yes

- name: Start Tomcat
  command: nohup /usr/local/tomcat/bin/startup.sh &
  sudo: yes

- name: Wait for port 8080
  wait_for: host=0.0.0.0 port=8080

- name: Download sample tomcat app
  get_url: url={{ tomcat_app }} dest=/usr/local/tomcat/webapps
```

The preceding screenshot shows the `tomcat` role that will uninstall the older version of Tomcat, download and install the newer version, and update the sample application. Let's see the role in greater detail.

We first remove the older version of Tomcat using the file module, which will make sure `usr/local/tomcat` does not exist, as shown in the following command lines:

```
- name: Remove older version of tomcat
  file: path=/usr/local/tomcat state=absent
```

We then download the new file archived in Tomcat using the `tomcat_url` variable passed to the role and the `get_url` module. Once we download the new version of Tomcat, we will extract and copy it to the `/usr/local/tomcat` directory. We've shown the deployment with just the `tar.gz` file. You could replace them with RPMs as well but, for this specific case where you have JAR or WAR files to deploy, we just went with the `tar.gz` file. However, if you wanted to enforce a standard way to deploy the `tar.gz` file across your organization, you could convert these `tar.gz` file to RPM. Let's look at the code for the explanation with `tar.gz`:

```
- name: Download Tomcat
  get_url: url={{ tomcat_url }} dest=/tmp


- name: Extract archive
```

```
  command: chdir=/tmp /bin/tar xvf apache-tomcat-{{ tomcat_version
}}.tar.gz


- name: Copy tomcat folder

  command: creates=/usr/local/tomcat mv /tmp/apache-tomcat-{{ tomcat_
version }}/ /usr/local/tomcat

  sudo: yes
```

We used the `command` module to extract the `tomcat` module. The `Extract archive` and `Copy tomcat folder` tasks can be replaced with a single task using the `unarchive` module that would look as follows:

```
- name: Extract archive

  unarchive: src=/tmp /bin/tar xvf apache-tomcat-{{ tomcat_version
}}.tar.gz dest=/usr/local/tomcat copy=no
```

With the plethora of modules available, there is always a tendency to use less optimized modules; always take a peek at the module list, however, and you might find something that will help you out! We'd like to recommend that you use as few tasks as possible to get things done. For this example, we'll stick to the two tasks that we originally wrote. Once we have the newer version of Tomcat installed, we will go ahead and start the Tomcat server and make sure that port `8080` is up and listening as follows:
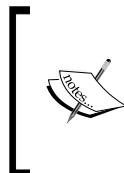
```
- name: Start Tomcat

  command: nohup /usr/local/tomcat/bin/startup.sh &

  sudo: yes


- name: Wait for port 8080

  wait_for: host=0.0.0.0 port=8080
```

> We use the `startup.sh` script with `nohup`, since Ansible will kill the `tomcat` process once it exits as it might end up being a foreground process. To keep the `tomcat` process running, we need to send it to the background. Again, if you remember what we said in the earlier part of the chapter, you could potentially replace this `startup.sh` script with a `/etc/init.d` script that you could directly call to start, stop, and restart your application.

Lastly, we will update the sample Tomcat app by using the `get_url` module. We use the `tomcat_app` variable to get the link from where we can download the `tomcat` sample application using the following command lines:

```
- name: Download sample tomcat app
  get_url: url={{ tomcat_app }} dest=/usr/local/tomcat/webapps
```

> You can enhance the tomcat playbook by using tags (to deploy the server and application, separately), `extra_variables` (to pass the Tomcat version), and the `unarchive` module. This is left up to you as an exercise. You can deploy a tomcat WAR file either before starting the service or after. We have done it post starting the service.

Let's run the preceding playbook and see how it works, as shown in the following screenshot:

```
$ ansible-playbook -i hosts playbooks/update_tomcat.yml

PLAY [Provisioning tomcat server] *****************************************

TASK: [Stopping monit] ****************************************************
changed: [54.227.33.114]

TASK: [Adding iptables rule to block port 8080] **************************
changed: [54.227.33.114]

TASK: [Saving iptables rule] *********************************************
changed: [54.227.33.114]

TASK: [tomcat | Remove older version of tomcat] *************************
changed: [54.227.33.114]

TASK: [tomcat | Download Tomcat] ****************************************
ok: [54.227.33.114]

TASK: [tomcat | Extract archive] *****************************************
changed: [54.227.33.114]

TASK: [tomcat | Copy tomcat folder] **************************************
changed: [54.227.33.114]

TASK: [tomcat | Start Tomcat] *******************************************
changed: [54.227.33.114]

TASK: [tomcat | Wait for port 8080] *************************************
ok: [54.227.33.114]

TASK: [tomcat | Download sample tomcat app] ****************************
changed: [54.227.33.114]

TASK: [Starting monit] ***************************************************
changed: [54.227.33.114]

TASK: [Flushing iptables rule] *******************************************
changed: [54.227.33.114]
```

So the run looks good. We truncated the output and pasted the final result, as shown in the following screenshot:

```
PLAY RECAP ********************************************************************
54.162.24.43                : ok=12    changed=10    unreachable=0    failed=0
54.162.98.221               : ok=12    changed=10    unreachable=0    failed=0
54.196.136.244              : ok=12    changed=10    unreachable=0    failed=0
54.227.33.114               : ok=12    changed=10    unreachable=0    failed=0
```

As you can see in the preceding Ansible run, Ansible updated the `tomcat` server and application on all the `tomcat` servers serially; at the end of it, you have a working deployment and, more importantly, a working application.

Deploying Ansible pull

The last topic we'd like to cover in this section is Ansible pull. If you have a large number of hosts that you'd like to release software on simultaneously, you will be limited by the number of parallel SSH connections that can be run. At scale, the pull model is preferred to the push model. Ansible supports what is called as **Ansible pull**. Ansible pull works individually on each node. The prerequisite is that it points to a repository from where it can invoke a special file called `localhost.yml` or `<hostname>.yml`. Typically, the `ansible-pull` option is run either as a cron job or is triggered remotely by some other means.

We're going to use our `tomcat` example again, with the only difference being that the structure of the repository has been changed slightly. Let's look at the structure of the git repository that will work for Ansible pull as follows:

```
$ tree
.
├── README.md
├── inventory
├── localhost.yml
├── roles
│   └── tomcat
│       └── tasks
│           └── main.yml
└── test

3 directories, 5 files
```

As you can see, `localhost.yml` is present at the top level and the `roles` folder consists of the `tomcat` folder, under which is the `tasks` folder with the `main.yml` task file. Let's now run the playbook using `ansible-pull` as follows:

```
[root@localhost ~]# ansible-pull -o -C master -d /opt/deployment -U https://github.com/madhurranjan/deployment.git -i localhost,
Starting ansible-pull at 2014-11-02 19:51:06

localhost | success >> {
    "after": "b3b604bbaf24566ae516830992ad2a4f84e28a98",
    "before": null,
    "changed": true
}


PLAY [Provisioning tomcat server] *********************************************

TASK: [Stopping monit] ********************************************************
changed: [localhost]

TASK: [Adding iptables rule to block port 8080] ******************************
changed: [localhost]

TASK: [Saving iptables rule] **************************************************
changed: [localhost]

TASK: [tomcat | Remove older version of tomcat] ******************************
changed: [localhost]

TASK: [tomcat | Download Tomcat] **********************************************
ok: [localhost]

TASK: [tomcat | Extract archive] **********************************************
changed: [localhost]

TASK: [tomcat | Copy tomcat folder] *******************************************
changed: [localhost]

TASK: [tomcat | Start Tomcat] *************************************************
changed: [localhost]

TASK: [tomcat | Wait for port 8080] *******************************************
ok: [localhost]
```

Let's look at the preceding run in detail as follows:

- The `ansible-pull` command: We invoke `ansible-pull` with the following options:
    - `-o`: This option means that the Ansible run takes place only if the remote repository has changes.
    - `-C master`: This option indicates which branch to refer to in the git repository.
    - `-U < >`: This option indicates the repository that needs to be checked out.
    - `-i localhost`: This option indicates the inventory that needs to be considered. In this case, since we're only concerned about one tomcat group, we use `-i localhost`. However, when there are many more inventory groups, make sure you use an inventory file with the `-i` option.

- The `localhost | success` JSON: This option checks whether the repository has changed and lists the latest commit.

- The actual Ansible playbook run: The Ansible playbook run is just like before. At the end of the run, we will have the WAR file up and running.

You don't need to use the `-o` option with `ansible-pull` if the playbooks are not going to change.

With this, we come to the end of the deployment and release aspect. It is recommended that you integrate the Ansible playbooks you write for deployment into a deployment pipeline using either ThoughtWorks Go (`http://www.thoughtworks.com/products/go-continuous-delivery`), Bamboo (`https://www.atlassian.com/software/bamboo`), or Jenkins (`http://jenkins-ci.org/`) along with its plugins to make sure you have specific deployment logs for each of your actual deployments; it is also recommended you add authorization to the pipelines to allow only specific teams to perform release deployments.

The following screenshot is an example of how you would run the same playbook but as a Jenkins job called `DeployTomcat`:

Finally, to end the chapter, you will succeed in a big way if you work towards making all your deployments for various applications a non-event!

Summary

We get it if you're a little tired at the end of this chapter. It did cover quite a few topics. To summarize, we looked at Deployment using a Rails application, Packaging and the importance of it, and Deployment strategies. You also learned about Canary and Rolling Deployments, Deployment Orchestration of a Tomcat application, and Ansible pull.

We've a few topics for you to think about, as follows:

- Do you use packaging for your applications? How do you perform releases today? How would you want to use Ansible, if possible, for performing releases?

- How do you handle configurations and database migrations for your applications? Can you use certain practices that we discussed here?

- Do you perform rolling or Canary deployments? If no, how would you lay it out using Ansible?

Finally, to end this book, we covered a whole gamut of topics around Ansible and how you can use Ansible as part of configuration management, deployments, alerting, and orchestration. We tried to provide as many examples as possible to illustrate the flow of tasks and we would be very happy if you take Ansible back to your organization and utilize what we've gone through in these chapters. We thoroughly enjoyed writing about Ansible and coming up with working samples. We hope you enjoyed reading the book as much as we did writing it. You might also find the *Appendix*, *Ansible on Windows, Ansible Galaxy, and Ansible Tower* useful but, for now, Adios! We'd like to end by saying:

"Besides black art, there is only automation (with Ansible!)"

-Modified quote of "Federico Garcia Lorca"

Ansible on Windows, Ansible Galaxy, and Ansible Tower

Ansible on Windows

Ansible Version 1.7 started supporting Windows with a few basic modules. There is no doubt that there will be extensive work around this area. Anyone who is familiar with Windows systems, understands Python, and likes Ansible, should wholeheartedly contribute to the Ansible-on-Windows effort. We'll briefly look at the setup process and try pinging a Windows system.

Let's look at the setup aspect as follows:

1. The command center or the control machine is a Linux machine. Ansible doesn't run on Cygwin. If you're a Windows shop, please make sure you have at least one Linux machine to control your environment.

    > The connection from the command center to all the machines is not over SSH; instead, it's over `winrm` or **Windows Remote Management** (**WinRM**). You can look up the Microsoft website for a detailed explanation and implementation at `http://msdn.microsoft.com/en-us/library/aa384426(v=vs.85).aspx`.

2. On the command center, once you've installed Ansible, it's important that you install `winrm`, which is compatible with Ansible, via pip, shown as follows:

    ```
    $ pip install https://github.com/diyan/pywinrm/archive/
    df049454a9309280866e0156805ccda12d71c93a.zip

    Downloading/unpacking https://github.com/diyan/pywinrm/archive/
    df049454a9309280866e0156805ccda12d71c93a.zip
    ```

```
   Downloading df049454a9309280866e0156805ccda12d71c93a.zip

   Running setup.py (path:/tmp/pip-bfPf0z-build/setup.py) egg_
info for package from https://github.com/diyan/pywinrm/archive/
df049454a9309280866e0156805ccda12d71c93a.zip
```

```
Requirement already satisfied (use --upgrade to upgrade):
xmltodict in /usr/lib/python2.6/site-packages (from
pywinrm==0.0.2dev)
```

```
Requirement already satisfied (use --upgrade to upgrade): isodate
in /usr/lib/python2.6/site-packages (from pywinrm==0.0.2dev)
```

```
Installing collected packages: pywinrm
```

```
   Running setup.py install for pywinrm
```

```
Successfully installed pywinrm
```

```
Cleaning up...
```

3. On each of the remote Windows machines, you need to install PowerShell Version 3.0. Ansible provides a couple of helpful scripts as follows to get you to set it up:

   ° Winrm Setup (`https://github.com/ansible/ansible/blob/devel/examples/scripts/ConfigureRemotingForAnsible.ps1`)

   ° An upgrade to PowerShell Version 3.0 (`https://github.com/cchurch/ansible/blob/devel/examples/scripts/upgrade_to_ps3.ps1`)

4. Allow `port 5986` via the firewall as this is the default WinRM connection port, and make sure this is accessible from the command center. To make sure you can access the service remotely, run a `curl` call: `curl -vk -d "" -u "$USER:$PASSWORD" "https://<IP>:5986/wsman"`. If Basic Auth works, you're set to start running commands.

5. Once the setup is done, you're now ready to start running Ansible! Let's run the equivalent of the Windows version of the "Hello, world!" program in Ansible by running `win_ping`. In order to do this, let's set up our credentials file. This can be done using Ansible Vault as follows:

```
$ ansible-vault  create group_vars/windows.yml
```

```
Vault password:
```

```
Confirm Vault password:
```

```
<Add content for the following Ansible variables>
```

```
ansible_ssh_user: Administrator
```

```
ansible_ssh_pass: <password>
```

```
ansible_ssh_port: 5986
```

```
ansible_connection: winrm
```

6. Let's set up our inventory file as follows:

```
$ cat inventory
[windows]
174.129.181.242
```

7. Followed by this, let's run `win_ping`:

```
$ ansible windows  -i inventory  -m win_ping --ask-vault-pass
Vault password:
174.129.181.242 | success >> {
    "changed": false,
    "ping": "pong"
}
```

We encourage you to start trying out the Windows modules present and keep an eye on Ansible for Windows as it's definitely headed in the right direction.

Ansible Galaxy

Ansible Galaxy is a free site where you can download Ansible roles developed by the community and kick-start your automation within minutes. You can share or review community roles so that others can easily find the most trusted roles on Ansible Galaxy. You can start using Ansible Galaxy by simply signing up with social media applications such as Twitter, Google, and GitHub, or by creating a new account on the Ansible Galaxy website (`https://galaxy.ansible.com/`) and downloading the needed roles using the `ansible-galaxy` command, which ships with Ansible Version 1.4.2 and later. Perform the following steps:

1. To download an Ansible role from Ansible Galaxy, use the following syntax:

   ```
   $ ansible-galaxy install username.rolename
   ```

   You can also specify a version as follows:

   ```
   $ ansible-galaxy install username.rolename[,version]
   ```

   If you don't specify a version, then `ansible-galaxy` will download the latest available version.

2. You can install multiple roles in two ways; firstly, by passing multiple role names separated by a space as follows:

```
$ ansible-galaxy install username.rolename[,version] username.
rolename[,version]
```

Secondly, you can do so by specifying role names in a file and passing that filename to the `-r/--role-file` option:

```
$ cat requirements.txt
user1.rolename,v1.0.0
user2.rolename,v1.1.0
user3.rolename,v1.2.1
```

3. You can then install roles by passing the filename to the `ansible-galaxy` command as follows:

```
$ ansible-galaxy install -r requirements.txt
```

4. Let's see how you can use `ansible-galaxy` to download a role for `apache httpd` as follows:

```
$ sudo ansible-galaxy install geerlingguy.apache
 downloading role 'apache', owned by geerlingguy
 no version specified, installing 1.2.3
 - downloading role from https://github.com/geerlingguy/ansible-role-apache/archive/1.2.3.tar.gz
 - extracting geerlingguy.apache to /etc/ansible/roles/geerlingguy.apache
geerlingguy.apache was installed successfully
```

5. The preceding `ansible-galaxy` command will download the `apache httpd` role inside the `/etc/ansible/roles` directory. You can now directly use the preceding role in your playbook, as follows:

```
$ cat playbooks/galaxy.yml
- hosts: web001
  remote_user: ec2-user
  sudo: yes
  roles:
    - { role: geerlingguy.apache }
```

> Make sure you have added /etc/ansible/roles to roles_path in your ansible.cfg file.

6. As you can see in the preceding screenshot, we created a simple playbook with a `geerlingguy.apache` role. This is the role that we downloaded using `ansible-galaxy`. Let's run this playbook and see how it works, as follows:

```
$ ansible-playbook -i hosts playbooks/galaxy.yml

PLAY [web001] ************************************************************

GATHERING FACTS *********************************************************
ok: [web001]

TASK: [geerlingguy.apache | Include OS-specific variables.] *******************
ok: [web001]

TASK: [geerlingguy.apache | Define apache_packages.] *************************
ok: [web001]

TASK: [geerlingguy.apache | Ensure Apache is installed.] **********************
changed: [web001] => (item=httpd,httpd-devel,mod_ssl,openssh)

TASK: [geerlingguy.apache | Update apt cache.] *******************************
skipping: [web001]

TASK: [geerlingguy.apache | Ensure Apache is installed.] **********************
skipping: [web001]

TASK: [geerlingguy.apache | Get installed version of Apache.] ****************
ok: [web001]
```

The preceding output is truncated because it's way too long to be pasted. The following screenshot is the final result of our playbook:

```
TASK: [geerlingguy.apache | Ensure Apache is started and enabled on boot.] ****
changed: [web001]

NOTIFIED: [geerlingguy.apache | restart apache] ******************************
changed: [web001]

PLAY RECAP **************************************************************
web001                     : ok=13    changed=5    unreachable=0    failed=0
```

As you can see, Ansible checked, installed, and configured the Apache `httpd` server on the `web001` host.

Ansible Tower

Ansible Tower is a web-based GUI developed by Ansible. Ansible Tower provides you with an easy-to-use dashboard to manage your nodes and role-based authentication to control access to your Ansible Tower dashboard. You can use LDAP or Active Directory to manage your users with Ansible Tower. Apart from the dashboard and role-based authentication, Ansible Tower also provides a built-in REST API, job scheduling, graphical inventory management, job status updates, and so on.

Ansible Tower is not freely available; you need to pay, depending on the number of nodes you want to manage. At the time of writing of this book, Ansible provided a free copy of Ansible Tower for 10 nodes. For more details, visit the Ansible Tower website at `http://www.ansible.com/tower` and the user guide is available at `http://releases.ansible.com/ansible-tower/docs/tower_user_guide-latest.pdf`.

Index

Symbols

--skip-tags
  about 85, 86
  install tag 86-88
  start tag 86-88
  stop tag 86-88
--syntax-check option
  using 75
-v option 29
-vv option 29
-vvv option 30

A

agent-based systems
  versus agentless systems 11
Alerting 169
Amazon Machine Image (AMI) 214
Amazon Web Services (AWS) 154
Ansible
  about 10-12
  configuring 20
  configuring, ansible.cfg used 21
  configuring, environment variables used 21
  functional testing 79
  Hello Ansible 17, 18
  installing 12
  installing, from source 13-15
  installing, package manager used 15, 16
  installing, via Apt 16
  installing, via Homebrew 16
  installing, via pip 16
  installing, via Yum 15
  need for 186
  on Windows 275-277

  repository, URL 194
  security 146
  URL 12
  URL, for inventory script 233
  used, for launching DigitalOcean
        instance 216-218
Ansible architecture 19, 20
ansible.cfg
  used, for configuring Ansible 21
ansible_connection parameter 47
Ansible Galaxy
  about 277-279
  URL 277
ansible-playbook command 38, 75
Ansible provisioner
  using, with Vagrant 71-74
ansible-pull command
  -C master option 271
  -i localhost option 271
  -o option 271
  about 271
  localhost | success JSON 272
Ansible pull deployment 270-272
ansible_python_interpreter parameter 47
ansible_shell_type parameter 47
ansible_ssh_host parameter 47
ansible_ssh_port parameter 47
ansible_ssh_private_key_file parameter 47
ansible_ssh_user parameter 47
Ansible Tower
  about 280
  URL 280
Ansible Vault
  using 146-148
Ansible workstation 12

Thank you for buying
Learning Ansible

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.
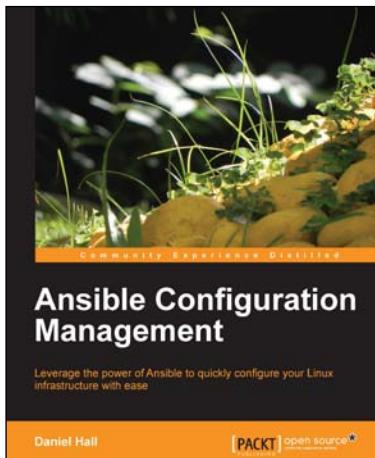
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
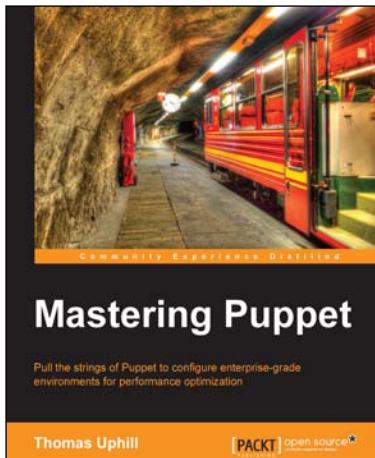
Ansible Configuration Management

ISBN: 978-1-78328-081-0   Paperback: 92 pages

Leverage the power of Ansible to quickly configure your Linux infrastructure with ease

1. Starts with the most simple usage of Ansible and builds on that.

2. Shows how to use Ansible to configure your Linux machines.

3. Teaches how to extend Ansible to add features you need.
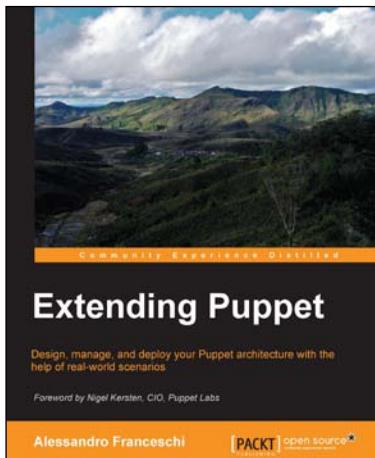
Mastering Puppet

ISBN: 978-1-78398-218-9   Paperback: 280 pages

Pull the strings of Puppet to configure enterprise-grade environments for performance optimization

1. Implement puppet in a medium to large installation.

2. Deal with issues found in larger deployments, such as scaling, and improving performance.

3. Step by step tutorial to utilize Puppet efficiently to have a fully functioning Puppet infrastructure in an enterprise-level environment.

Please check **www.PacktPub.com** for information on our titles
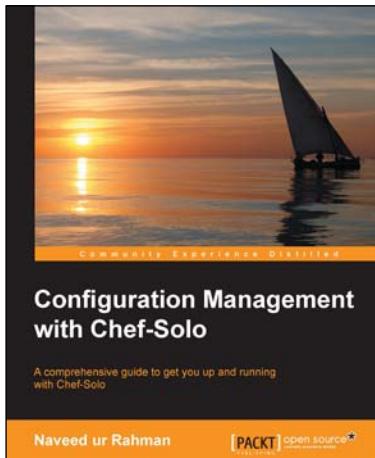
Extending Puppet

ISBN: 978-1-78398-144-1   Paperback: 328 pages

Design, manage, and deploy your Puppet architecture with the help of real-world scenarios

1. Plan, test, and execute your Puppet deployments.

2. Write reusable and maintainable Puppet code.

3. Handle challenges that might arise in upcoming versions of Puppet.

Configuration Management with Chef-Solo

ISBN: 978-1-78398-246-2   Paperback: 116 pages

A comprehensive guide to get you up and running with Chef-Solo

1. Explore various techniques that will help you save time in infrastructure management.

2. Use the power of Chef-Solo to run your servers and configure and deploy applications in an automated manner.

3. This book will help you to understand the need for the configuration management tool and provides you with a step-by-step guide to maintain your existing infrastructure.

Please check **www.PacktPub.com** for information on our titles