Community Experience Distilled

Creating Development Environments with Vagrant

Create and manage virtual development environments with Puppet, Chef, and VirtualBox using Vagrant

Michael Peacock

[PACKT] PUBLISHING open source* community experience distilled

Creating Development Environments with Vagrant

Create and manage virtual development environments with Puppet, Chef, and VirtualBox using Vagrant

Michael Peacock

BIRMINGHAM - MUMBAI

Creating Development Environments with Vagrant

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Production Reference: 1200813

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-918-2

www.packtpub.com

Cover Image by Neha Rajappan (neha.rajappan1@gmail.com)

Credits

Author
Michael Peacock

Reviewer
Chad Thompson

Acquisition Editor
Owen Roberts

Commissioning Editor
Manasi Pandire

Technical Editors
Manal Pednekar
Larissa Pinto

Project Coordinator
Akash Poojary

Proofreader
Paul Hindle

Indexer
Mariammal Chettiyar

Production Coordinator
Adonia Jones

Cover Work
Adonia Jones

About the Author

Michael Peacock (`www.michaelpeacock.co.uk`) is an experienced Senior/Lead Developer and a Zend Certified Engineer from Newcastle, UK, with a degree in Software Engineering from the University of Durham.

After spending a number of years running his own web agency, managing the development team, and working for Smith Electric Vehicles on developing their web-based Vehicle Telematics platform, he currently serves as a CTO for Ground Six (`www.groundsix.com`), an ambitious tech company, where he leads the development team and manages the software development processes.

He is the author of *Drupal 7 Social Networking*, *PHP 5 Social Networking*, *PHP 5 E-Commerce Development*, *Drupal 6 Social Networking*, *Selling Online with Drupal E-Commerce*, and *Building Websites with TYPO3*. Other publications Michael has been involved in include *Mobile Web Development*, *Drupal for Education and E-Learning*, and *Jenkins Continuous Integration Cookbook*, for which he acted as a Technical Reviewer.

Michael has also presented at a number of user groups and conferences including PHP UK Conference, Dutch PHP Conference, ConFoo, PHPNE, PHPNW, and Cloud Connect.

You can follow Michael on Twitter, `@michaelpeacock`, or find out more about him through his blog, `www.michaelpeacock.co.uk`.

I'd like to thank all the staff at Packt Publishing, in particular, Erol Staveley, Robin de Jongh, Akash Poojary, and Manasi Pandire for seeing this book through to fruition. I'd also like to thank my Technical Reviewer, Chad Thompson, who helped ensure the technical quality of the book was up to scratch.

My thanks also go to my friends and family, in particular, my wife Emma for her support while working on the book.

Finally, I'd like to thank you, the reader; I hope you enjoy this book and enjoy the benefits of using virtualized development environments with Vagrant!

About the Reviewer

Chad Thompson is a software developer, architect, and builder in central Iowa, and is currently employed as a DevOps Engineer with Dice Holdings, Inc. in Urbandale, IA. Chad has many years of experience in creating and helping others create great technology, from working closely with development teams to speaking and writing. Chad is currently serving as a Senior Contributing Author for the SELECT Journal published by the Independent Oracle Users Group. He has also written articles for a number of online publications and spoken at many industry conferences and events. You can find other writings, presentations, and more information about Chad at `http://chadthompson.me`.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across 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 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 nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface 1

Chapter 1: Getting Started with Vagrant 5

Requirements for Vagrant 8

Getting installed 8

Installing VirtualBox 8

Installing Vagrant 13

Summary 14

Chapter 2: Managing Vagrant Boxes and Projects 15

Creating Vagrant projects 15

Importing and using base boxes 16

Creating projects without importing a base box 19

Managing Vagrant boxes 20

Adding Vagrant boxes 21

Listing Vagrant boxes 22

Removing Vagrant boxes 22

Repackaging Vagrant boxes 23

Finding Vagrant boxes 23

Controlling guest machines 23

Powering up the virtual machine 23

Suspending the virtual machine 25

Resuming the virtual machine 25

Shutting down the virtual machine 25

Starting from scratch 26

Connecting to the virtual machine over SSH 26

Integration between the host and the guest 27

Port forwarding 27

Synced folders 27

Networking 28

Auto-running commands 28
Summary 29
Chapter 3: Provisioning with Puppet 31
Provisioning 32
About Puppet 32
Creating modules and manifests with Puppet 33
Puppet classes 33
Default Puppet manifests 34
Resources 35
Resource execution ordering 37
Installing software 37
Updating our package manager 38
Installing the Apache package 38
Running the Apache service 39
File management 39
Copying a file 39
Creating a symlink 40
Creating folders 41
Creating multiple folders in one go 41
cron management 42
Running commands 42
Manage users and groups 43
Creating groups 43
Creating users 43
Updating the sudoers file 44
Subscribe and refresh only 44
Puppet modules 45
Using Puppet to provision servers 45
Summary 46
Chapter 4: Provisioning with Chef 47
Knowing about Chef 48
Creating cookbooks and recipes with Chef 48
Resources – what Chef can do 49
Installing software 49
Updating our package manager 50
Installing the Apache package 50
Running the Apache service 51
Understanding file management 51
Copying a file 51
Creating a symlink 52
Creating folders 53
Creating multiple folders in a single process with looping 53
Managing cron 54
Running commands 54

Managing users and groups 55
Creating groups 55
Creating users 55
Updating the sudoers file 56
Knowing common resource functionalities 56
Using Chef cookbooks 56
Using Chef to provision servers 57
Summary 57
Chapter 5: Provisioning with Vagrant using Puppet and Chef 59
Provisioning within Vagrant 59
Provisioning with Puppet on Vagrant 60
Using Puppet in a standalone mode 60
Puppet provisioning in action 61
Using Puppet in client/server mode 62
Provisioning with Chef on Vagrant 62
Using Chef solo 63
Using Chef in client/server mode 64
Other built-in provisioners 64
Provisioning with SSH – a recap 65
Ansible playbooks 65
Using multiple provisioners on a single project 65
Overriding provisioning via the command line 66
Summary 67
Chapter 6: Working with Multiple Machines 69
Using multiple machines with Vagrant 70
Defining multiple virtual machines 70
Connecting to multiple virtual machines over SSH 71
Networking multiple virtual machines 72
Provisioning the machines separately 74
Destroying a multi-machine project 75
Summary 75
Chapter 7: Creating Your Own Box 77
Getting started 77
Preparing the VirtualBox machine 78
VirtualBox Guest Additions 83
Vagrant authentication 84
Vagrant user and admin group 84
Sudoers file 85
Insecure public/private key pair 85

Provisioners **86**
Puppet 86
Chef 86
Cleanup **87**
Export **87**
Summary **87**
Appendix: A Sample LAMP Stack **89**
Index **99**

Preface

Web-based software projects are increasingly complicated, with a range of different dependencies, requirements, and interlinking components. Swapping between projects, which require different versions of the same software, becomes troublesome. Getting team members up and running on new projects also becomes time-consuming.

Vagrant is a powerful tool for creating, managing, and working with virtualized development environments for your projects. By creating a virtual environment for each project, their dependencies and requirements are isolated, and don't interfere; they also don't interfere with software installed on your own machine such as WAMP or MAMP. Colleagues can be up and running on a new project in minutes with a single command. With Vagrant, we can wipe the slate clean if we break our environment, and be back up and running in no time.

What this book covers

Chapter 1, *Getting Started with Vagrant*, introduces the concept of virtualization, its importance in the role of the development environment, and walks through the Vagrant installation process.

Chapter 2, *Managing Vagrant Boxes and Projects*, walks through creating Vagrant projects, exploring and configuring the Vagrantfile, and working with base boxes.

Chapter 3, *Provisioning with Puppet*, explores the provisioning tool Puppet and how to create Puppet manifests to provision a server.

Chapter 4, *Provisioning with Chef*, explores the provisioning tool Chef and how to create Chef recipes to provision a server.

Chapter 5, *Provisioning with Vagrant using Puppet and Chef*, discusses how to use both Puppet and Chef within the context of Vagrant to provision development environments.

Chapter 6, *Working with Multiple Machines*, explores using Vagrant to create and manage projects, which use multiple virtual machines which communicate with each other.

Chapter 7, *Creating Your Own Box*, discusses the process of creating your own base box for use within a Vagrant project.

Appendix, *A Sample LAMP Stack*, walks through the process of creating a LAMP server within a new Vagrant project.

What you need for this book

You will need a Windows, OS X, or Linux computer with Vagrant and Oracle's VirtualBox installed, although the install process for these will be discussed in *Chapter 1*, *Getting Started with Vagrant*.

Who this book is for

This book is for software developers, development managers, and technical team leaders who want to have a more efficient, robust, and flexible development environment for their projects and for their team.

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: "New team members can be onboarded to new projects as easy as `git clone && vagrant up`".

A block of code is set as follows:

```
class apache {
  package { "apache2":
    ensure => present,
    require => Exec['apt-get update']
  }
}
```

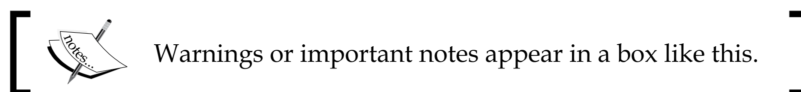
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
server1.vm.provision :puppet do |puppet|
  puppet.manifests_path = "provision/manifests"
  puppet.manifest_file  = "server1.pp"
  puppet.module_path = "provision/modules"
end
```

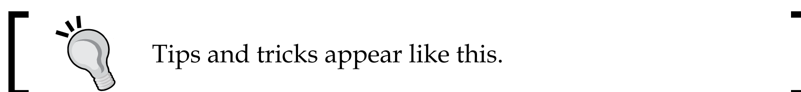
Any command-line input or output is written as follows:

```
vagrant init precise64 http://files.vagrantup.com/precise64.box
```

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: "Again, on OS X, the first step is to double-click on the **Vagrant.pkg** icon".



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 Vagrant

Developing web-based applications can be complex. We have to be able to work with teams of people, who all need to be able to run and work on these projects, and we work with an ever-changing stack of technologies. I know personally I've spent countless hours setting up developers onto new projects in the past, and countless hours wrangling with WAMP and MAMP to switch to a newer or older version of PHP when juggling multiple projects. With everyone in a team working on their own machine, with their own development tools on their own operating systems, it's almost impossible to keep a consistent configuration across all the machines; especially if you have remote workers or freelancers where you can't force IT policies on them.

As projects get more complicated, it's also easier for auxiliary configurations to be forgotten about. Asynchronous workers, message queues, cron jobs; typically, we need to tell the rest of the team about these, and hope someone remembers them when it is time to deploy.

A virtualized development environment can help with this. Instead of having to battle configurations when working on other projects, each project can simply have its own virtualized environment. It can have its own dedicated web server, database server, and the versions of the programming language and other dependencies it needs. Because it is virtualized, it doesn't impact on other projects, just shut it down and boot up the environment for the other project.

With a virtualized environment, the development environments can also mimic the production environment. No more needing to worry if something will work when it gets deployed, if it is being developed on a machine with the exact same software configuration. Even if you deploy on a Linux machine but develop on Windows, your virtualized environment can be Linux, running the same distribution as your production environment.

While a virtualized environment makes things easier, by their nature, they are not the easiest of things to configure and manage themselves. They still need to be configured to work with the project in question, which often involves some level of system administrator skills, and we need to connect to these environments and work with them. They also, by design, are not very portable. You need to export a large image of the virtualized environment and share that with colleagues, and keeping that up-to-date as projects evolve can be cumbersome. Thankfully, there is a tool, which can manage these virtualized environments for us and provide a simple interface to configure them; an interface which involves storing configuration in simple plain text files, which are easy to share with colleagues, keeping everyone up-to-date as the project changes. This tool is **Vagrant**.

Vagrant (`http://www.vagrantup.com/`) is a powerful development tool, which lets you manage and support the virtualization of your development environment. Instead of running all your projects locally on your own computer, having to juggle the different requirements and dependencies of each project, Vagrant lets you run each project in its own dedicated virtual environment.

Vagrant provides a command-line interface and a common configuration language, which allows you to easily define and control virtual machines which run on your own systems, but which tightly integrate, allowing you to define how your own machine and the virtual machine interact. This can involve syncing folders such that the project code on your computer, which you edit using your IDE is synced so that it runs on the Vagrant development environment.

Vagrant uses **Providers** to integrate with the third-party virtualization software, which provides the virtualized machines for our development environment. The default provider is for Oracle's **VirtualBox** however, there are providers to work with Amazon Web Services and VMware Fusion. The entire configuration is stored in simple plain text files. The Vagrant configuration (**Vagrantfile**), **Puppet**, and **Chef** manifests are simply written in text files in a Ruby Domain Specific Language. This means we can easily share the configurations and projects with colleagues, using Version Control Systems such as Git or Subversion.

When using Vagrant, the next time you need to go back to a previous project, you don't need to worry about any potential conflicts with changes made to your development environment (for example, if you have upgraded PHP, MySQL, or Apache on your local environment, or within the Vagrant environment for another project). If you bring a new team member into the team, they can be up and running in minutes Vagrant will take care of all the software and services needed to run the project on their machine. If you have one project, which uses one web server such as Apache, and another which uses Nginx, Vagrant lets you run these projects independently.

If your project's production environment involves multiple servers (perhaps one for the Web and one for the database), Vagrant lets you emulate that with separate virtual servers on your machine.

With Vagrant:

- Your development environment can mimic the production environment.
- Integrated provisioning tools such as Puppet and Chef allow you to store configuration in a standard format, which can also be used to update production environments.
- Each project is separate in its own virtualized environment, so issues as a result of configuration and version differences for dependencies on different projects are things of the past.
- New team members can be onboarded to new projects as easy as `git clone && vagrant up`.
- "It works on my machine" is an excuse of the past.
- The headache of linking code that you write on your own machine to your virtualized development environment, is taken care of either through custom-synced folders or the default-synced folder (everything in your project's folder gets mapped to Vagrant).
- The environment can act as if it was your local machine and can map the web server port (80) of your development machine to your development environment if you wish.
- You can let colleagues view your own development environment, as well as easily share the development environment.
- Your local WAMP or MAMP installations will be gathering dust!

In this chapter, we will:

- Discuss the requirements and prerequisites for Vagrant
- Install Oracle's VirtualBox
- Install Vagrant
- Verify if Vagrant was successfully installed

Once we have Vagrant and its prerequisites on our machine, we can then look at using it for our first project.

Requirements for Vagrant

Vagrant can be installed on Linux, Windows, and Mac OS X, and although it uses Ruby, the package includes an embedded Ruby interpreter. The only other requirement is a virtualization tool such as Oracle's VirtualBox. The Oracle's VirtualBox provider is available for free, and is included built-in with Vagrant, so we will use and install VirtualBox in order to use Vagrant during the course of this book. Other providers are available, including one for VMware Fusion or Workstation, which is available as a commercial add-on (`http://www.vagrantup.com/vmware`).

Getting installed

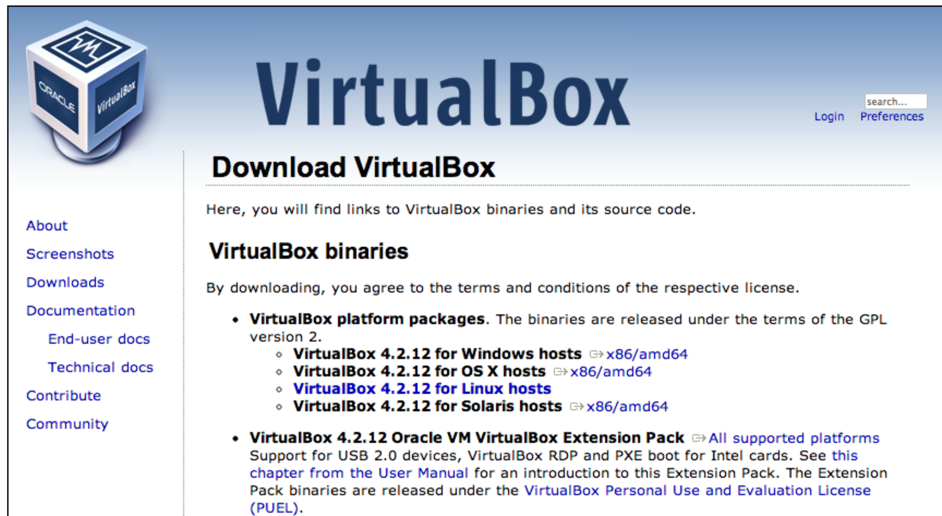
Now that we know the software, which we need in order to get Vagrant running on our machine, let's start installing VirtualBox (so that we can use Vagrant's built-in VirtualBox provider) and Vagrant itself.

Installing VirtualBox

VirtualBox (`https://www.virtualbox.org/`) is an open source tool sponsored by Oracle, which lets you create, manage, and use virtual machines on your own computer.

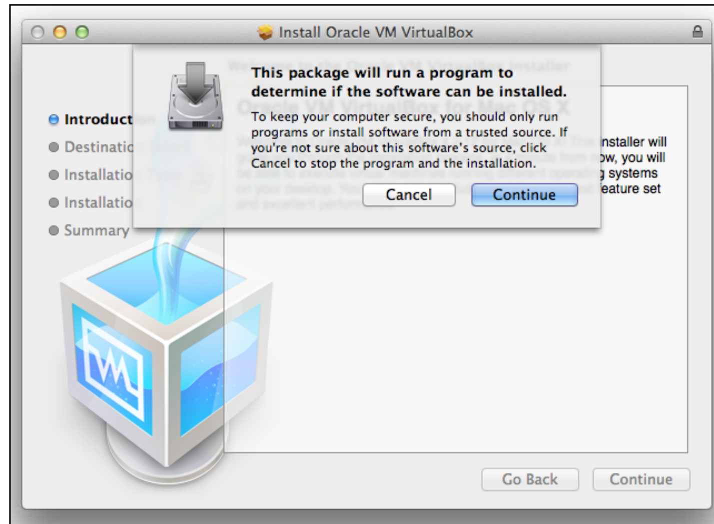
VirtualBox is a graphical program, which lets you visually create virtual machines, allocate resources, load external media such as operating system CDs, and view the screen of the virtual machine. Vagrant wraps on top of this and provides an intuitive command-line interface along with integration of additional tools (including provisioners such as Puppet and Chef), so that we don't need to worry about how VirtualBox works or what to do with it; Vagrant takes care of it for us.

The first stage is to download the installer from the VirtualBox downloads page (`https://www.virtualbox.org/wiki/Downloads`). We need to select the download, which relates to our computer (OS X, Windows, Linux, or Solaris).



Once downloaded, let's open it up and run the installer. On OS X, this involves clicking on the **VirtualBox.pkg** icon that is shown on the screen. On Windows, simply opening the installer opens the installation wizard.

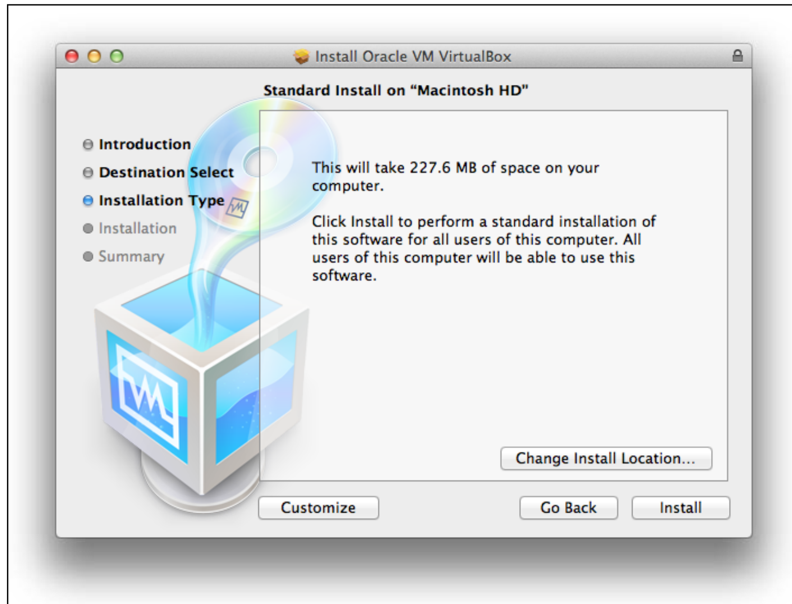
Before the installer runs, it first checks to see if the computer is capable of having VirtualBox installed we need to click on **Continue** to begin the installation process. While this process will vary from OS X to Windows to Linux, the process is very similar across all platforms. There are fully detailed installation instructions for all platforms on the VirtualBox website (`https://www.virtualbox.org/manual/ch02.html`).



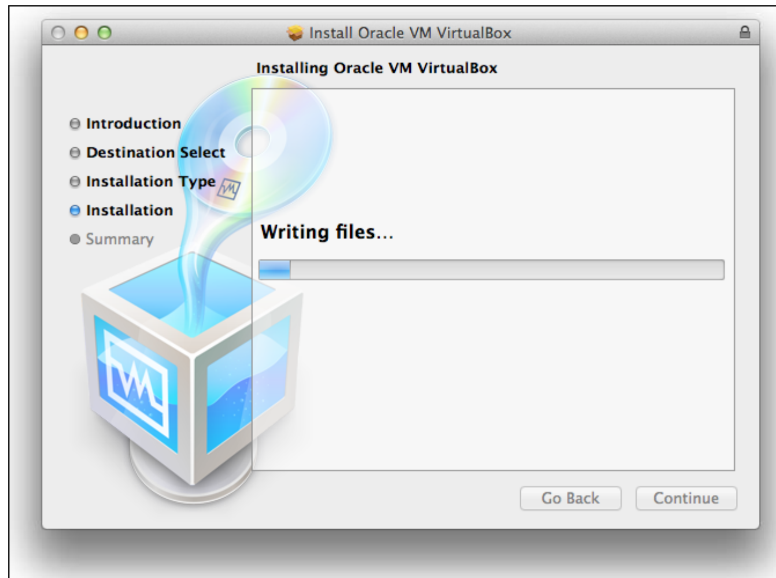
The first step in the process provides us with an introduction to the installation process and reminds us as to what we are actually installing.

Next, the installer informs us as to how much space it will use on our computer, and provides us with the option to customize the installation if we want to **Change Install Location...** and install the software in another location (perhaps another disk drive if our disk is getting full).

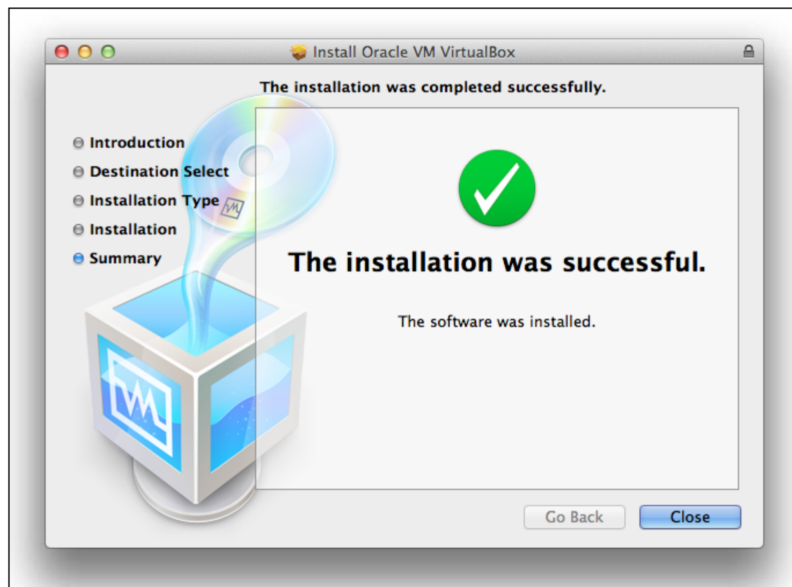
Let's leave the default install location as it is, and click on the **Install** button to have the installer install VirtualBox on our computer.



The installer then automatically installs VirtualBox for us.



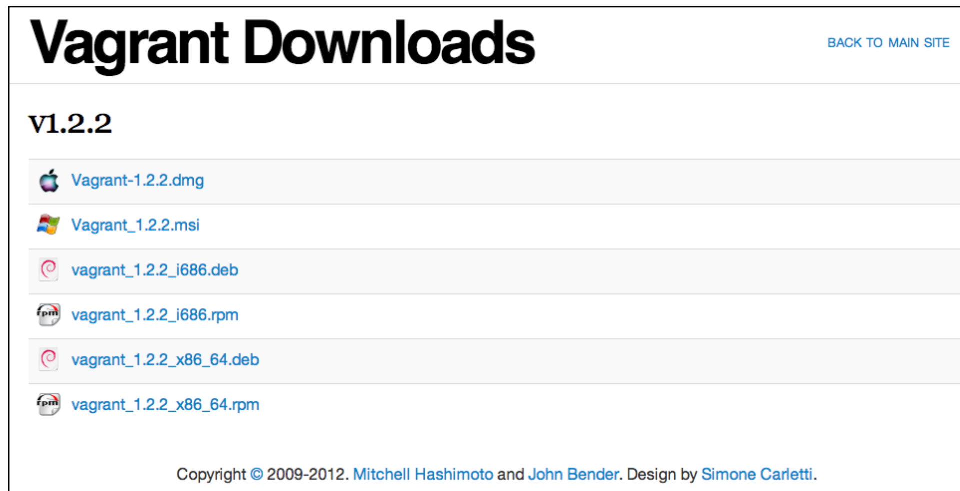
Once the installation has finished, we are shown a confirmation screen with the option of clicking on **Close** to close the installer.



Now we have successfully installed VirtualBox!

Installing Vagrant

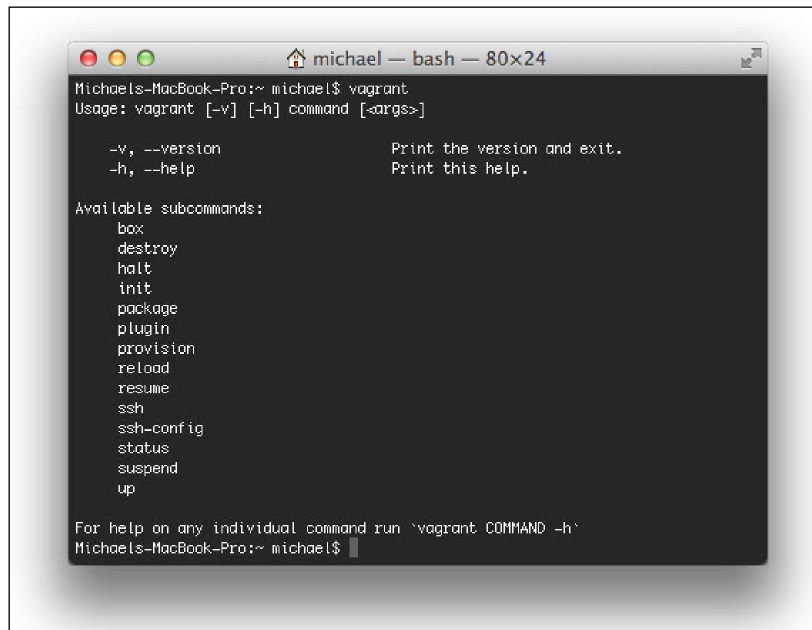
Now that we have the prerequisites installed on our computer, we can actually install Vagrant itself. This process is similar to that of installing VirtualBox. First, let's download the relevant installer from the Vagrant page (`http://downloads.vagrantup.com/tags/v1.2.2`).



Let's open up the installer and start the process. Again, on OS X, the first step is to double-click on the **Vagrant.pkg** icon.

We now need to follow the installation steps which are provided; this is very similar to the earlier steps for VirtualBox, and for most of the software packages in general.

Let's verify if Vagrant has been successfully installed, by opening a command prompt (terminal on Linux/OS X or cmd on Windows) and running Vagrant.



The preceding output shows that we have successfully installed Vagrant, and that we are able to run it.

Summary

In this chapter, we have looked at the requirements and prerequisites for Vagrant, which include a virtualization tool such as Oracle's VirtualBox (which works with Vagrant's built-in VirtualBox provider). We then downloaded and installed both Oracle's VirtualBox and Vagrant, and ran Vagrant to check if it was installed correctly.

Now that we have it installed, we can now move on to using Vagrant to set up and manage some of our projects.

2

Managing Vagrant Boxes and Projects

In this chapter, we will learn the basics of using Vagrant. We will look at initializing projects, importing base boxes to be used as our operating system, and controlling the virtual machine by powering on and off, suspending and resuming, and connecting to the box. Finally, we will also learn how to configure some of the key integration points between our own machine and our Vagrant-controlled virtual machine, including:

- Port forwarding
- Folder mapping
- Networking

Creating Vagrant projects

Now that we have Vagrant installed on our machine, let's look at creating projects. There are three different ways we can do this:

- Create a new project with a named base box and a location where the box can be downloaded if we don't already have it setup
- Create a new project with a named base box
- Create a new project, which will get the default base box name

Let's look at how we can do this.

> While we are going to look at commands to initialize our Vagrant projects in this chapter, these are simply quick ways to create a `Vagrantfile` file with some values pre-populated. `Vagrantfile` is the configuration file, which defines how Vagrant should use the project (operating system to be used, virtual machines to boot up, synced folders, forwarded ports, and so on).

Importing and using base boxes

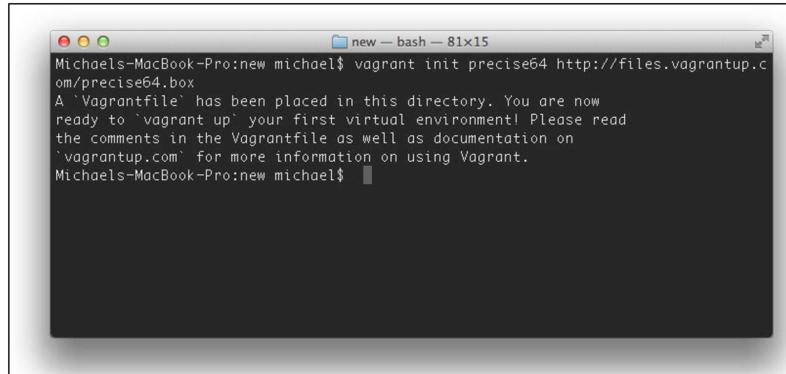
Each virtual machine starts with what Vagrant calls a base box. This is a specially packaged version of an operating system with some specific configurations in place. The number of configurations and packages installed on this packaged operating system is typically minimal (containing only a few tools which allow it to communicate with Vagrant). It is the job of the end user to install the additional software on our virtual machines, using provisioning tools. We will look at provisioning later in this book, which will automate the process of taking a base Vagrant box and converting it into an environment suitable for our project, for example, by installing software such as a web server and a database server, and configuring the appropriate programming languages.

Provided we are in the directory we wish to convert into a new Vagrant project, we can simply run the following command at the terminal:

```
vagrant init precise64 http://files.vagrantup.com/precise64.box
```

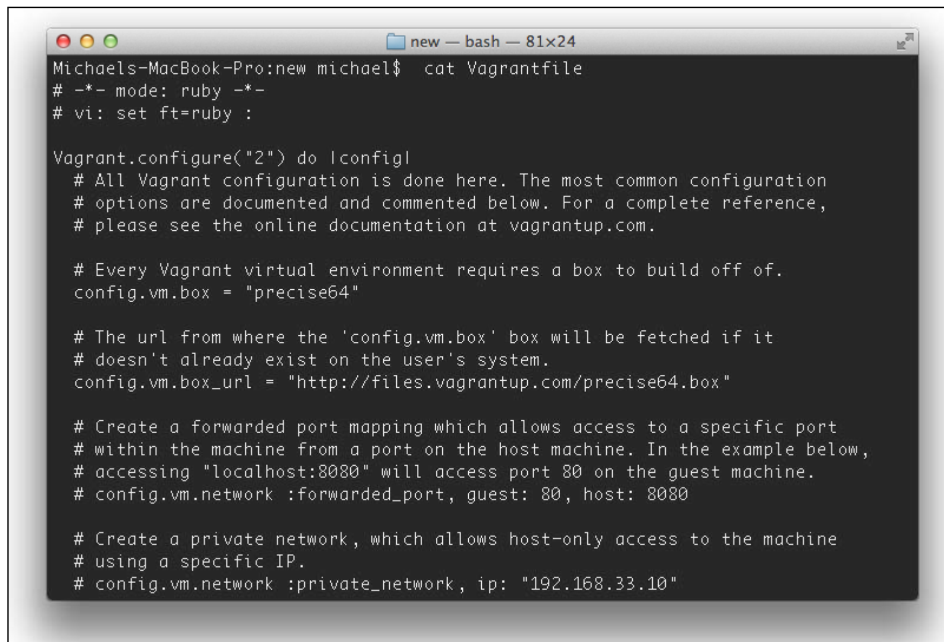
This runs the `init` subcommand within Vagrant, and instructs Vagrant to create a new project with configuration to use the box named `precise64`, and if the box is not found, to import the box located at `http://files.vagrantup.com/precise64.box` when the Vagrant environment is booted for the first time. The name `precise64` can be used within other new and existing projects to refer to this base box. Base boxes are downloaded and stored in a place Vagrant can access and reuse.

After a base box is downloaded, the screen will look as follows:

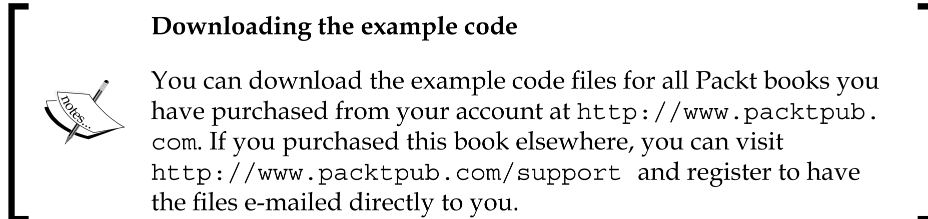


The initialization of the new project creates a file named `Vagrantfile` within our project's folder. When we go to boot a Vagrant virtual environment, Vagrant looks for this configuration file to determine what to do. Because everything related to the Vagrant environment is either within this file or the provisioning (SSH, Puppet, and Chef) files within our project, it's easy to maintain the environment under version control and share it with colleagues.

Let's open up the file named `Vagrantfile` and take a look inside:



The Vagrant configuration file is written in Ruby, and the default `Vagrantfile` we get is primarily comments illustrating some of the ways we can customize the file. These comments are prefixed with the # character.



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.

There are only four lines which are not comments within the file; let's look at them now.

The first line tells Vagrant that this is Version 2 of the configuration object; Version 2 of the configuration object is, somewhat confusingly, designed for Vagrant Versions 1.1 through to 2.0:

```
Vagrant.configure("2") do |config|
```

This next line instructs Vagrant to use the box named `precise64`:

```
config.vm.box = "precise64"
```

If we are working with a new Vagrant project, which uses a base box we haven't installed, we need a mechanism for Vagrant to find and download the base box for us. This next line tells Vagrant where it can download a copy of the box named previously (`precise64`). When the VM is booted for the first time, if the base box is not found, then it will be downloaded from the URL provided. We will see this in action once we boot the VM for the first time in the *Controlling guest machines* section. This can either be a web address or a path to a file on the filesystem or network:

```
config.vm.box_url = "http://files.vagrantup.com/precise64.box"
```

Finally, we tell Vagrant that the configuration has ended, and it should stop processing the configuration:

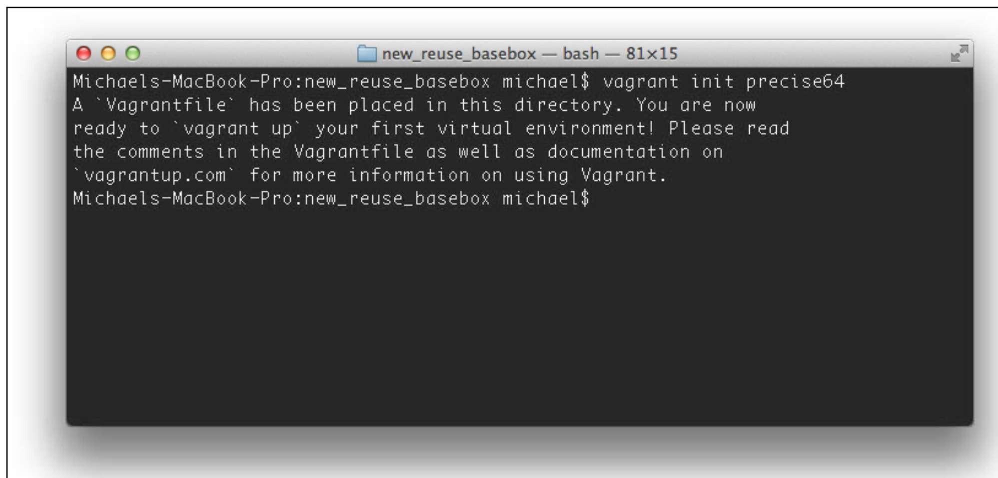
```
end
```

Creating projects without importing a base box

Of course, we only need to import a base box if we need to create a project using a base box we have not already installed on our system. Once a box has been imported, it is usable within new and existing Vagrant projects, using the name it was imported with. So, let's create a new project utilizing the `precise64` base box we imported earlier:

```
vagrant init base64
```

As before, it then creates a new `Vagrantfile` and the project is set up ready for us:



The difference between initializing a project this way and the way we did earlier is that Vagrant isn't aware of the fallback URL for the base box location. Even if we use a base box that is already imported to Vagrant, it doesn't provide a fallback URL. If you are going to share the project with colleagues, be sure to add a fallback URL so that they can grab a copy of the base box.

We can also create a new Vagrant project with the following command:

```
vagrant init
```

This will again create `Vagrantfile`, but with two exceptions to the preceding:

- It won't have a fallback URL
- The base box name that the project will use will be `base`

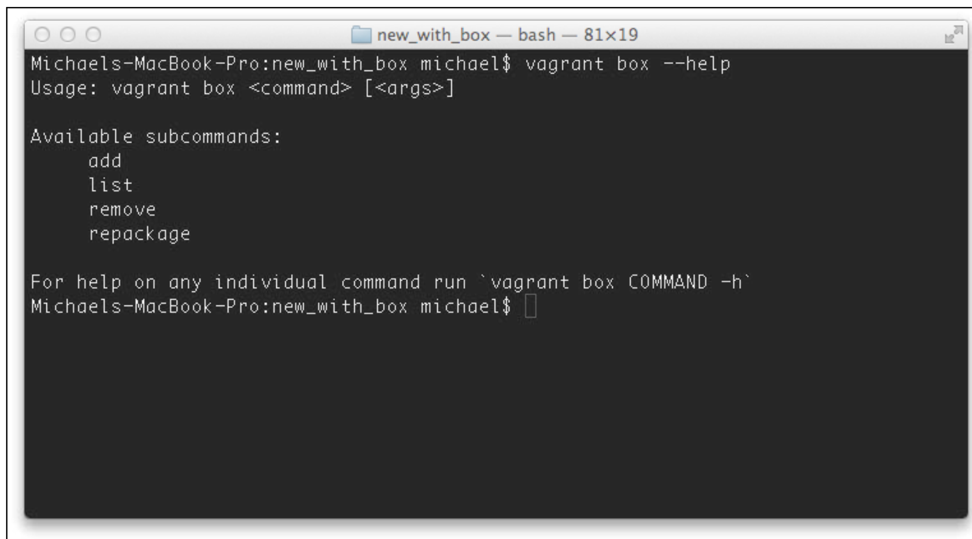
We can of course edit the `Vagrantfile` to provide an alternative base box name if we wish, or we can ensure that we have a base box setup called `base` within Vagrant.

Managing Vagrant boxes

We can manage Vagrant boxes using the `vagrant box` subcommands. Let's run that with the help flag (`--help`) and see what subcommands are available:

```
vagrant box --help
```

The following screenshot shows all the available subcommands:



There are four available box-related subcommands. With each of these, we can provide the `--help` flag to see what additional arguments are available. The available box-related subcommands are:

```
vagrant box add <name> <url> [--provider provider] [--force]
vagrant box list
vagrant box remove <name> <provider>
vagrant box repackage <name> <provider>
```

Let's review these in detail.

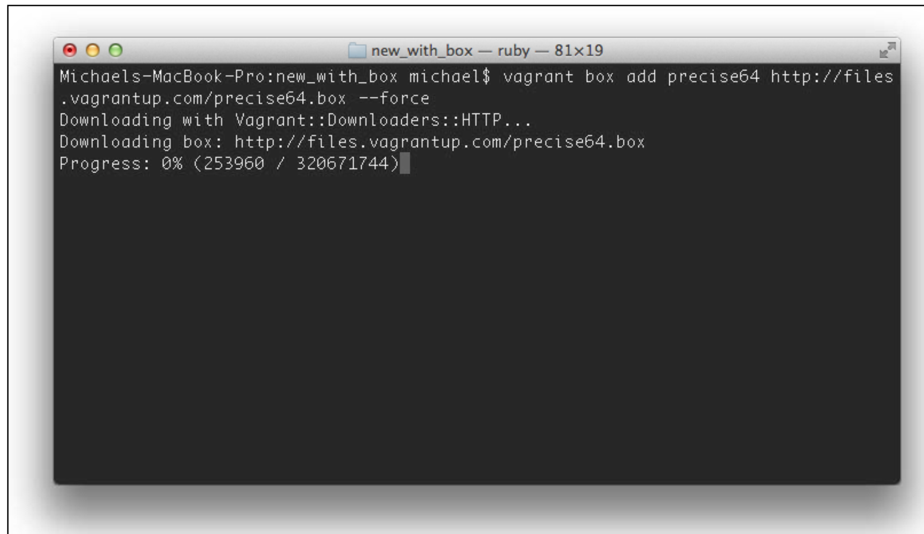
Adding Vagrant boxes

The `add` subcommand allows us to add a new box. It has two arguments and two optional flags. The arguments are the `name` parameter of the box and the `URL` parameter to download it from. The optional flags are `--force`, which will tell vagrant to remove a pre-existing box with the same name, and `--provider`, which will allow us to specify another provider to back the box (the default provider being VirtualBox; however, there are providers available for Vagrant including VMware and Amazon S3).

The following command would add a new `precise64` box, and if an existing one is found, it will override it:

```
vagrant box add precise64 http://files.vagrantup.com/precise64.box --force
```

The following screenshot depicts adding a new `precise64` box:



This process may take a while, as most Vagrant boxes will be at least 200 MB big. Once downloaded, the box will be extracted and available for us to use in our Vagrant projects:

```
Michaels-MacBook-Pro:new_with_box michael$ vagrant box add precise64 http://files
.vagrantup.com/precise64.box --force
Downloading with Vagrant::Downloaders::HTTP...
Downloading box: http://files.vagrantup.com/precise64.box
Extracting box...
Cleaning up downloaded box...
Successfully added box 'precise64' with provider 'virtualbox'!
Michaels-MacBook-Pro:new_with_box michael$
```

Listing Vagrant boxes

The `list` subcommand will list the boxes installed within Vagrant along with the provider that backs the box:

```
vagrant box list
```

The following screenshot shows all the boxes available within Vagrant:

```
Michaels-MacBook-Pro:new_with_box michael$ vagrant box list
lucid32            (virtualbox)
precise64          (virtualbox)
quantal64_roderik  (virtualbox)
Michaels-MacBook-Pro:new_with_box michael$
```

Removing Vagrant boxes

We can remove the box with the `remove` subcommand. We need to provide the name of the box and the provider that backs it. For example:

```
vagrant box remove lucid32 virtualbox
```

Let's see it in action:

```
Michaels-MacBook-Pro:new_with_box michael$ vagrant box remove lucid32 virtualbox
Removing box 'lucid32' with provider 'virtualbox'...
Michaels-MacBook-Pro:new_with_box michael$
```

Repackaging Vagrant boxes

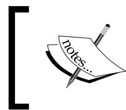
The `repackage` subcommand lets us convert a Vagrant environment, complete with any customizations we have made to it, such as software we have installed on it, into a box which we can reuse and distribute to others. We will use this command in *Chapter 7*, *Creating Your Own Box*.

Finding Vagrant boxes

The Vagrant project provides a few boxes, which we can use at the time of writing; these are currently Ubuntu Lucid and Ubuntu Precise, both as 32-bit and 64-bit installs, as these are Long Term Support editions:

- Lucid32 is available at `http://files.vagrantup.com/lucid32.box`
- Lucid64 is available at `http://files.vagrantup.com/lucid64.box`
- Precise32 is available at `http://files.vagrantup.com/precise32.box`
- Precise64 is available at `http://files.vagrantup.com/precise64.box`

We may, however, want to use another operating system or Linux distribution. One option would be to create our own box, which we will discuss in *Chapter 7*, *Creating Your Own Box*, but we can also find and download them from other sources. One such source is the website `vagrantbox.es`. This is a website listing various boxes that others have setup and provided for download.



Be careful using any box you download from an untrusted source, as you can't guarantee what software is or isn't running on there without further investigation.

Controlling guest machines

Now that we have a project initialized, we need to be able to control our guest machine. At the moment, all we have is a `Vagrantfile` file, which defines the configuration for the project.

Powering up the virtual machine

We can power up the virtual machine using the `vagrant up` command.

Firstly, Vagrant checks to see if the Vagrant environment has already been set up, if a previously suspended environment is found, it will resume it.

If the environment was not previously suspended, Vagrant then checks to see if the base box has already been downloaded onto the machine. If it hasn't, it will download it.

Vagrant will then perform the following:

- Copy the base box
- Create a new virtual machine with the relevant provider (the default being VirtualBox)
- Forward any configured ports; by default, it will forward port 22 (SSH) on the VM to port 2222 on the host; this will allow us to connect to the VM
- Boot (power up) the VM
- Configure and enable networking, so that we can communicate with the VM
- Map shared folders between the host and the guest (by default, it will map the folder containing the Vagrant project to `/vagrant` on the guest machine)
- Run any provisioning tools that are set up such as Puppet, Chef, or SSH provisioning

After powering up the virtual machine, the screen looks something like as follows:

```
Michaels-MacBook-Pro:new_with_box michael$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
[default] Importing base box 'precise64'...
[default] Matching MAC address for NAT networking...
[default] Setting the name of the VM...
[default] Clearing any previously set forwarded ports...
[default] Fixed port collision for 22 => 2222. Now on port 2200.
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Preparing network interfaces based on configuration...
[default] Forwarding ports...
[default] -- 22 => 2200 (adapter 1)
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Configuring and enabling network interfaces...
[default] Mounting shared folders...
[default] -- /vagrant
Michaels-MacBook-Pro:new_with_box michael$
```

Suspending the virtual machine

We can save the current state of the virtual machine to disk so that we can resume it later. If we run `vagrant suspend`, it will suspend the VM and stop it from consuming our machine's resources (except for the disk space it will occupy), ready for us to resume later:

```
Michaels-MacBook-Pro:new_with_box michael$ vagrant suspend
[default] Saving VM state and suspending execution...
Michaels-MacBook-Pro:new_with_box michael$
```

Resuming the virtual machine

In order to resume a previously suspended virtual machine, we simply run `vagrant resume`:

```
Michaels-MacBook-Pro:new_with_box michael$ vagrant resume
[default] Resuming suspended VM...
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
Michaels-MacBook-Pro:new_with_box michael$
```

Shutting down the virtual machine

We can shut down a running virtual machine using the `vagrant halt` command. This instructs the VM to stop all running processes and shut down. To use it again, we need to run `vagrant up`, which will power on the machine; by default, the `up` command will re-run any provisioning tools we have set up:

```
Michaels-MacBook-Pro:new_with_box michael$ vagrant halt
[default] Attempting graceful shutdown of VM...
Michaels-MacBook-Pro:new_with_box michael$
```

Starting from scratch

Sometimes, things go wrong. It's not inconceivable that we might make some changes to our virtual machine and find out that it no longer works. Thankfully, since we have a base box, configuration file, and provisioning files, which are all stored separately, we can instruct Vagrant to destroy our virtual machine, and then create it again, using the configurations to set it up. This is done via the `destroy` subcommand, and then the `up` subcommand to start it again:

```
vagrant destroy
vagrant up
```

Of course, if we update our `Vagrantfile`, provisioning manifests, or application code that can also break things; so it is important we use a Version Control System to properly manage our project's code and configuration so we can undo changes there to; Vagrant can only do so much to help us!

Connecting to the virtual machine over SSH

If we run the `vagrant ssh` command, Vagrant will then connect to the VM over SSH. Alternatively, we could use SSH to connect to localhost with port 2222, and this will tunnel into the VM.

Let's see it in action:

```
Michaels-MacBook-Pro:new_with_box michael$ vagrant ssh
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:23:18 2012 from 10.0.2.2
vagrant@precise64:~$
```

If we are running Vagrant on a Windows machine, we won't have a built-in SSH client. We can use a client such as **PuTTY** to connect to Vagrant. PuTTY can be downloaded from `http://www.chiark.greenend.org.uk/~sgtatham/putty/`. More information is available on the Vagrant website for configuring PuTTY to work with Vagrant (`http://docs-v1.vagrantup.com/v1/docs/getting-started/ssh.html`).

Integration between the host and the guest

Without any form of integration between the host machine and the guest, we would have a virtual server running on top of our own operating system, which is not particularly useful. We need our own machine to be capable of integrating tightly with the guest (virtual machine).

Port forwarding

Although the VM is running on our own machine, it acts and behaves like a completely different machine. Sometimes this is what we want, other times it isn't. We may want to integrate the two more tightly. One option is port forwarding, where we can tunnel a port from the virtual machine to a port on the host machine. If, for example, we have a web server running on our own machine, we obviously don't want to map the web server port from Vagrant onto the same port, otherwise there would be a conflict. Instead, we can map it to another port. If we map the web server port on the VM to port 8888 on the host, then visiting `http://localhost:8888` on our own machine would show us the web service we run on the guest.

Port forwarding is done via lines in the `Vagrantfile` file, we simply provide the guest and host ports we wish to map:

```
config.vm.network :forwarded_port, guest: 80, host: 8888
```

If we have other Vagrant-managed virtual machines on our computer, which we wish to run simultaneously, we can enable `auto_correct` on specific ports; this way, if a conflict is found (for example, two VMs trying to map to the same port), one will try a different port instead:

```
, auto_correct: true
```

Ports below a certain range need elevated/root privileges on the host machine, so you may be asked for your administrative password.

Synced folders

Synced folders allow us to share a folder between the host and the guest. By default, Vagrant shares the folder containing the Vagrant project as `/vagrant` on the VM. We can use the following in our `Vagrantfile` to sync more folders if we wish:

```
config.vm.synced_folder "/Users/michael/assets/"
"/var/w
ww/assets"
```

The first parameter is the path to the folder on our machine, the second being the mount point on the VM. If we use a relative path on our machine, it would be relative to the project folder.

The Network File System can give us better performance with synced folders than the default settings, this won't work on Windows hosts, and on Linux/OS X hosts will require root privileges. We can enable the NFS on a per synced folder basis by adding the following to the preceding line:

```
, :nfs => true
```

Networking

By default, our Vagrant virtual machines are only accessible from the machines we run Vagrant on. If we map ports to our host, then we can share services running on the VM with our colleagues within our network. If we want to allow our colleagues to access our Vagrant-managed VMs directly, we can attach the VM to our internal network, and VirtualBox will bridge the network between our machine and the VM and the internal network between our machine and the rest of the machines in your home or office:

```
config.vm.network :private_network, ip: "192.168.1.100"
```

Auto-running commands

One of the key concepts within Vagrant is provisioning. This involves turning a basic virtual machine with a base operating system into a server that is ready to run for your project, meeting your requirements. There are three key provisioning options within Vagrant:

- SSH
- Puppet
- Chef

Puppet and Chef are both third-party tools which Vagrant supports out-of-the-box, and provide specific languages for configuring servers in an agnostic way that can be used for different operating systems. The next two chapters will discuss these in more detail.

SSH provisioning involves running a series of commands on the virtual machine over SSH when the VM is first setup.

There are two ways we can use SSH provisioning. We can either directly run a command from our `Vagrantfile` file with the following line:

```
config.vm.provision :shell, :inline => "sudo apt-get update"
```

Alternatively, we can tell Vagrant to run a particular shell script (the location of the script specified is relative to our project root, that is, `/vagrant`):

```
config.vm.provision :shell, :path => "provision.sh"
```

This shell script could contain all of the commands we need to convert a base box, which contains very little, to a box which supports our project and application, perhaps installing web and database servers.

Summary

In this chapter, we created projects with Vagrant, pulling in Vagrant boxes to use. We also looked at the commands needed to manage these boxes and to manage the Vagrant virtual machines. We looked at how we can configure our Vagrant environment with networking, synced folders, and forwarded ports. Finally, we looked at how to provision software on our VM with SSH commands.

3
Provisioning with Puppet

Vagrant is a very powerful tool because of the two concepts it can manage for us:

- Virtualization
- Provisioning

We have already learned through *Chapter 1, Getting Started with Vagrant* and *Chapter 2, Managing Vagrant Boxes and Projects* how to use Vagrant to manage virtual machines for us, and this is very useful. However, at this stage, these virtual machines are dumb; they have very little software installed for us to use, and they are certainly not configured for our projects. This is where provisioning comes in. Provisioning automates the process of turning a base machine into one, which is configured for use with a specific project.

In this chapter, we are going to quickly look through the basics of **Puppet**, one of the three main provisioning options available within Vagrant. We won't look at it within a Vagrant context just yet; we will simply look at how a Puppet works, and how we can use it. In *Chapter 5, Provisioning with Vagrant using Puppet and Chef*, we will look at how to connect what we will learn in this chapter with Vagrant itself. In this chapter, we will learn:

- How the Puppet works
- The basics behind the Puppet modules and manifests
- How to use Puppet to perform the following tasks:
    - Install software
    - Manage files and folders within the filesystem
    - Manage `cron` jobs
    - Run commands
    - Manage users and groups

- How to use third party Puppet modules and **Puppet Forge**
- How to manually run Puppet to provision a machine

Puppet itself is a large topic, and subject of several books. For a more detailed look at it, Packt have some titles dedicated to it:

- *Puppet 2.7 Cookbook*, `http://www.packtpub.com/puppet-2-7-for-reliable-secure-systems-cloud-computing-cookbook/book`
- *Puppet 3 Beginner's Guide*, `http://www.packtpub.com/puppet-3-beginners-guide/book`

Provisioning

Within this context, Provisioning is the process of setting up a virtual machine so that it can be used for a specific purpose or project. Typically, this involves installing software, configuring the software, managing services running on the machine, and even setting up users and groups on the machine.

For a web-based software project, provisioning will likely entail the installation of a web server, a programming language, and a database system. Configuration changes will be needed to set up a database on the database system and to allow the web server to write to specific folders (to deal with user uploads).

Without this provisioning process, we would have an almost vanilla install of an operating system which contains a synced copy of our project folder; this vanilla install wouldn't be usable as a development environment for our project. Provisioning takes us to the next level and gives us a fully working environment for our project.

About Puppet

Puppet is a provisioning tool which we can use to set up a server for use for a project. The configuration which determines how the server needs to be set up can be stored within our Vagrant project and can be shared with teammates through a version control, ensuring everyone gets an up-to-date copy of the required development environment.

Information about how a server should be configured, that is, its software, files, users, and groups, is written into files known as the Puppet manifests. These manifests are written by using Puppet's own language, which is a **Ruby Domain Specific Language**. Puppet takes this information and compiles it into a catalog that is specific for the operating system it is being applied to. The catalog is then applied to the machine.

For our purposes we will be using Puppet in standalone mode (this is also how Vagrant uses it). Standalone mode means that everything runs from one machine. Puppet also has client-server capabilities, where you can define the Puppet manifests for all the servers in your infrastructure, on a central host, and it keeps your individual servers at the required level of configuration.

Puppet is idempotent, which means running Puppet on a machine multiple times has the same effect as running it only once. In effect Puppet ensures that conditions are met, and if they are not, it will perform actions to ensure they are, for example, if we were to ensure Apache was installed, Puppet would install it if it was not already installed, if it was already installed it would do nothing. This means we can re-provision with Puppet many times without any detrimental effect. This is useful as we can use it to keep the server in sync with our Puppet manifests if they were to change.

Creating modules and manifests with Puppet

Puppet is made up of a manifest file, and a number of modules (which also contain manifests and other resources). The default manifest specifies which modules are to be used, and depending on the module, provides customization options to it (for example, the Puppet module for supervisord allows us to specify any number of processes which should be managed by using supervisord through the module itself).

Modules make use of resources within Puppet to control and configure the machine, and these modules can be imported to run in a specific sequence, through stages.

Puppet classes

Puppet modules typically consist of classes, which in turn utilize a number of resource types (in this example, the package resource type, to install a software package) to achieve a specific requirement for our server. It effectively allows us to bundle a number of these resource types in a way which means we can simply include the class by its name and have all of the instructions from within it executed.

A class in its most basic form is structured as follows:

```
class apache {
  package { "apache2":
    ensure => present,
    require => Exec['apt-get update']
  }
}
```

For its most basic use within Vagrant, classes like this would be saved as `default.pp` within the `modules/apache/manifests/` folder. The class can contain many resource types to achieve a desired goal (for instance, installing the Apache package isn't the same as preparing the web server fully for a project, related tasks can be bundled into the same class).

Default Puppet manifests

For a given project, Puppet modules are typically all located in a specific modules directory. Many modules can be customized when they are run, an example being the supervisord module, it simply provides the structure for us to customize for each process we want it to manage.

Because of this we need to have a default Puppet manifest that includes a list of modules to be run, and any configurations for them. Because Puppet is aware of our module folder location when we run it (and when it is run through Vagrant) we just list the modules to include and run.

A basic manifest, which would include and run the Apache class we wrote earlier, would be:

```
import "apache"
include apache
```

I've mentioned the supervisord module (`https://github.com/plathrop/puppet-module-supervisor`) a few times as a module which is designed to be used for multiple different things which the developer using it can customize. Supervisord is a tool which maintains a number of running processes, for example, if you have a background worker in a web application to resize images, supervisor might be responsible for keeping five workers running at any one instance, re-spawning them when one has finished. The following is an example of how this module would be used in a default Puppet manifest:

```
supervisor::service {
  'resize_images':
    ensure      => present,
    enable      => true,
    command     => '/usr/bin/php /vagrant/src/private/index.php
      img_resize',
    user        => 'root',
    group       => 'root',
    autorestart => true,
    startsecs => 0,
```

```
        num_procs => 5,
        require     => [ Package['php5-cli'], Package['beanstalkd'] ];
    }

    supervisor::service {
      email':
        ensure      => present,
        enable      => true,
        command     => '/usr/bin/php /vagrant/src/private/index.php
          email',
        user        => 'root',
        group       => 'root',
        autorestart => true,
        startsecs => 0,
        num_procs => 5,
        require     => [ Package['php5-cli'], Package['beanstalkd'] ];
    }
```

Here we are instructing Puppet to use the supervisord module twice, to set up and manage two workers for us. For each of the workers we have a set of five processes to run, and we have to set the user and group to run them as, and we have defined PHP's command line interface and the beanstalkd worker queue as requirements for the workers. This illustrates the power that Puppet modules have.

Resources

Puppet provides a range of resource types which we can utilize to create our configuration files. These resource types are translated and compiled depending on the operating system being used. For example, if we were to use the `package` resource type to install some software, this would use `apt-get` on Ubuntu and Yum on Fedora operating systems. A small number of resource types are operating system specific, for example, the `scheduled_task` resource type is designed specifically for Windows, and the `cron` type is designed for Linux and UNIX based systems.

Resource types available include:

- `cron`: To manage `cron` jobs on Linux and UNIX based systems
- `exec`: To run commands at the terminal/command prompt
- `file`: To manage and manipulate files and folders on the filesystem
- `group`: To manage user groups
- `package`: To install software
- `service`: To manage running services on the machine
- `user`: To manage user accounts on the machine

When resource types are used directly (for example, we use the `Package` resource type to install some software) they are used in lower case (`package`), however, when we refer to a resource type we have used, for example, as a requirement for another Puppet action, we reference them with a leading capital letter (`Package`).

An example of this is as follows:

```
package { "apache2":     ensure => present,   require =>
  Exec['apt-get update'] }
```

We are telling Puppet to install the `Apache2` package (lower case p for `package`), but when we specify the requirement of a previously executed `exec` command, we use a leading capital letter. The options within this instruction for Puppet (`ensure` and `require` keywords) are called **parameters**.

A full list of resource types is available on the Puppet website: `http://docs.puppetlabs.com/references/latest/type.html`.

When using a resource type, a name is provided (in the preceding instance, this is `Apache2`) this is often dual purpose, serving both as a way for us to reference the action (in this case, the package being installed) and also as an instruction (in this case, what package Puppet needs to install). When it comes to the `Exec` resource type, the name is the command we wish to run, by default we need to provide the full path to the command we run, we can avoid that by providing the path from which the command should be run as a parameter.

Resource requirements

Certain things we do with Puppet will require other actions to have been performed first. These can be defined by using the `require` parameter, and we can specify multiple requirements.

If we need to run or install something after both the MySQL Server and the MySQL Client packages have been installed, we would use the `require` parameter to define these as follows:

```
require => [ Package['mysql-client'], Package['mysql-server'] ]
```

The outer square brackets are used because we are defining multiple requirements, so we wrap them in square brackets and separate them with commas.

Resource execution ordering

Sometimes, we need to run specific blocks of the Puppet code before other blocks. In most cases, we can use the `require` parameter to get around this problem. However, if we have one `exec` command requiring other `exec` commands, it starts to get a little hairy.

Puppet has a default stage within which all commands run. We can create our own stages which run before or after this stage, which allows us to force commands to be run in specific orders.

We can define stages within our default puppet manifest and then instruct Puppet to run certain classes from within that stage. If, for example, we wanted to run our `Apache` class before anything else, we could create a stage to run first and put the `Apache` class within that stage as follows:

```
stage { 'first': before => Stage[main] }
class {'apache': stage => first}
```

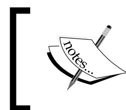
This creates a stage called `first`, anything assigned to this stage will be executed before the default Puppet stage; next it places the `Apache` class within that stage. Once we have a named stage such as `first` we can then create other stages which can run before this one too.

Installing software

Let's say we want to install Apache on our server. There are three typical steps involved in this process:

1. Update our package manager
2. Install the Apache package
3. Run the Apache service

Because the first step is different depending on the operating system we are running, we might want to move this out of Puppet at a later stage; however, we will use it within Puppet for the time being. Any operating specific commands (such as this) are written for Ubuntu, which is the operating system we are using with Vagrant. If you are not using Ubuntu, the `exec` command should be re-written to update the package manager on your system.



This example is purely to illustrate the process of putting together a simple module. There are many existing modules available on Puppet Forge, which we will come to later.

Updating our package manager

In order to update our package manager, we need to run a command on the server. This can be achieved by using the `Exec` resource within Puppet:

```
exec {  'apt-get update':
    command => '/usr/bin/apt-get update',
    timeout => 0
}
```

This instructs Puppet to run the `apt-get update` command. We have set a timeout of zero so that if it takes a while (and after a fresh install of an operating system through Vagrant, it might), Puppet will run it for as long as it takes, overriding the default timeout.

This isn't the most elegant of approaches, especially with it being operating system specific and subsequently a requirement for most of our other commands. In *Appendix A*, we will build a **LAMP** server project with Vagrant and Puppet, and in the example there we use Vagrant's SSH provisioning options to update the package manager before we install the other software. Most base boxes don't have up-to-date package details to save space and due to their age, so updating the package manager is required.

Installing the Apache package

We can use the `Package` resource to ensure that Apache is installed, and if it isn't, it will be installed, as follows:

```
package { "apache2":
   ensure => present,
      require => Exec['apt-get update']
}
```

Here we have told Puppet to ensure that the `Apache2` package is present. We have added our `apt-get update` command as a prerequisite, so we know that our packages will be up-to-date.

Running the Apache service

Finally, make sure that Apache uses the `Service` resource. While Apache will automatically run when we install it, we may connect to our new server and alter settings or services by mistake. If this happens, we can simply re-run the provisioner; as Apache will already be installed it won't re-install it, but the `Service` resource will force Puppet to ensure that the Apache service is running. Obviously, this can't be run if Apache isn't installed, so the `Apache2` package is a prerequisite.

```
service { "apache2":
    ensure => running,
      require => Package['apache2']
}
```

File management

We can use the `File` resource within Puppet to manage files and folders within the filesystem. Let us look at some examples, which allow us to:

- Copy files
- Create symlinks
- Create folders
- Create multiple folders in one go

Copying a file

One common file operation we might want to perform would be to take a configuration file from our project and copy it into our virtual machine. One particular use case would be an Apache configuration file; we might want to define some virtual hosts and other settings in a file which we can share with our colleagues.

While this works well and can get us up and running quickly, there are modules out there which allow us to configure Apache and other software directly from Puppet. This typically works by the module storing a template file (a copy of the configuration file with placeholders in it) and then inserting data we define within Puppet into the template and copying the file onto the machine. However, for the sake of this introductory chapter we will just copy a file across.

The `file` resource type allows us to create files, folders, and symlinks. In order to create a file (or replace the contents of an existing file with another file), we simply tell Puppet the file we want to create or replace (the destination), the source (that is, the file to copy and put into the destination), and the user and group who should own the file:

```
file { '/etc/apache2/sites-available/default':
  source => '/vagrant/provision/modules/apache/files/default',
    owner => 'root',
      group => 'root',
require => Package['apache2']
}
```

As this is an Apache configuration file, it is worth ensuring that Apache is already installed, otherwise Apache would override this when it installs the first time and wouldn't make the process idempotent.

> A note about file locations: The source file in the above `file` resource code is held within a Vagrant environment within the Puppet module itself. We can provide two kinds of file path, either the full path to the file on the machine which Puppet is running (our Vagrant environment) or a path relative to Puppet modules. These Puppet paths are structured as: `puppet:///modules/apache/default`. The difference you will note is that it automatically looks in the files/ folder within the Apache folder, we don't need to specify that.

Creating a symlink

If we omit the `source` parameter and instead add an `ensure` parameter and set that to link, we can create a symlink. A target is used to define where the symlink should point to, as shown in the following code:

```
file { '/var/www/vendor':
    ensure => 'link',
      target => '/vagrant/vendor',
        require => Package['apache2']
}
```

Creating folders

Similar to the preceding symlink code, this time we simply need to set `ensure` to a directory. This will then create a directory for us, as follows:

```
file{ "/var/www/uploads":
    ensure => "directory",
      owner  => "www-data",
        group  => "www-data""
          mode   => 777,
}
```

We can use the `mode` parameter to set the permissions of the folder (this also can be used for files we create and manage).

Creating multiple folders in one go

In many web projects we may need to create a number of folders within our servers or our Vagrant virtual machines. In particular, we may want to create a number of cache folders for different parts of our application, or we may want to create some upload folders.

In order to do this, we can simply create an array containing all of the folders we want to create. We can then use the `file` resource type and pass the array to create them all as follows:

```
$cache_directories = [ "/var/www/cache/", "/var/www/cache/pages",
  "/var/www/cache/routes", "/var/www/cache/templates",
]

file { $cache_directories:
  ensure => "directory",
    owner  => "w"w-data",
      group  => "www-data",
        mode   => 777,
}
```

cron management

The cron resource type lets us use Puppet to manage cron jobs which we need to run on the machine. We provide a name, in this case web_cron, the command to run, the user to run the command as, and the times at which to run the command, as shown in the following code:

```
cron { web_cron:
    command  => "/usr/bin/php /vagrant/cron.php",
      user  => "root",
        hour  => [1-4],
          minute  => [0,30],
}
```

Puppet provides us with different configuration options to define the times at which a cron should be run, which include:

- Hour: between 0 and 23 inclusive
- Minute: between 0 and 59 inclusive
- Month: between 1 and 12 inclusive
- Monthday: between 1 and 31 inclusive
- Weekday: Sunday (7 or 0) to Saturday (6)

If one of these is omitted from the configuration, then Puppet runs it for every one of the available options (that is, if we omit month, it will run for January through to December). When defining the dates and times, we can either provide a range, for example, [1-5], or specifics, such as [1, 2, 10, 12].

Running commands

The Exec resource type allows us to run commands through the terminal on the machine we are provisioning. One caveat with the exec command is that if you re-provision with Puppet it will re-run the command, which depending on the command could be damaging. What we can do with the Exec resource type is set the creates parameter. The creates parameter tells Puppet of a file that will be created when the command is run, if Puppet finds that file, it knows that it has already been run and won't run it again.

Take for example, the following configuration, this would use the PHP composer tool to download dependencies. The command itself creates a file called `composer.lock` (we could, of course, use the `exec` command itself to create a file manually, perhaps by using the `touch` command). Because of the lock file that is created, we can use the `creates` parameter to prevent the command from being executed if it has previously been executed and has created the lock file, as shown in the following code:

```
exec{ "compose":
  command => '/bin/rm -rfv /var/www/repo/vendor/* && /bin/rm -f
    /var/www/repo/composer.lock && /usr/bin/curl -s
      http://getcomposer.org/installer | /usr/bin/php && cd
        /var/www/repo && /usr/bin/php /var/www/repo/composer.
          phar install',
  require => [ Package['curl'], Package['git-core'] ],
    creates => "/var/www/repo/composer.lock",
      timeout => 0
}
```

Manage users and groups

The `user` and `group` resource types let us create and manage users and groups. There are differences between different operating systems as to what Puppet can do with the users and groups and how this works. The following code is tested on Ubuntu, Linux. More information on the differences for users and groups on different platforms can be found on the Puppet website: `http://docs.puppetlabs.com/references/latest/type.html#user`.

Creating groups

The simplest way to create a group is simply to set the `ensure` parameter to `present`.

```
group { "wheel":
  ensure => "present",
}
```

Creating users

To create a user, the basic information we should provide is:

- The username
- The fact we want the user to exist (`ensure => present`)
- The group (`gid`) the user should be part of
- The shell to use for the user

- The home directory for the user
- If we want Puppet to manage the home directory for the user. In this case it will create the folder for us if it does not exist
- The password for the user
- The requirement that we need the `wheel` group in place first

The code which will then create our user is as follows:

```
user { "developer":
  ensure => "present",
  gid => "wheel",
  shell => "/bin/bash",
  home => "/home/developer",
  managehome => true,
  password => "passwordtest",
  require => Group["wheel"]
}
```

Updating the sudoers file

It's all well and good being able to create users and groups on our machine, however, one thing that we can't do by using the `user` and `group` resource types is define a user or group as having elevated privileges, unless we make the user a part of the `root` group.

We can use an `exec` command to push some text to the end of our `sudoers` file; the text we need to push just tells the file that we want to give the `wheel` group the sudo privileges, as shown in the following code:

```
exec { "/bin/echo \"%wheel  ALL=(ALL) ALL\" >> /etc/sudoers":
  require => Group["wheel"]
}
```

Subscribe and refresh only

Sometimes we want to have a Puppet command run multiple times when other commands have finished. One example is restarting Apache. We would want to do this:

- When we import a new configuration file
- When we install Apache modules such as PHP support and `mod_rewrite`
- If we add new virtual hosts

This can be achieved by using the `subscribe` and `refreshonly` parameters. The `subscribe` parameter instructs the command to run every time any of the commands in the **subscribe** option have been run.

The `refreshonly` parameter, when set to `true`, instructs the command to only run when one of the commands it subscribes to, has run (leaving this out would mean the command is also run in its own right):

```
exec { "reload-apache2":
  command => "/etc/init.d/apache2 reload",
  refreshonly => true,
  subscribe => File['/etc/apache2/sites-available/default'],
}
```

Here the command to reload Apache will only be run when the new configuration file has been loaded. We can, of course, extend the `subscribe` parameter to contain other things such as modules and other configurations, as discussed.

Puppet modules

There are many existing, well written, re-usable Puppet modules freely available to use. Puppet Forge is a collection of these, available at `http://forge.puppetlabs.com/`. It is always worth looking to see if there is an existing module which solves your problem before writing your own.

Using Puppet to provision servers

We are going to look at how to use Puppet with Vagrant in *Chapter 5*, *Provisioning with Vagrant using Puppet and Chef*, however, Puppet can also be run independently. Provided Puppet is installed (it will be on most Vagrant base boxes, but if you want to run it on another machine, it might not be, so install it first), you can use the `apply` subcommand, passing with it the location of the modules folder and the default manifest to apply, as follows:

```
puppet apply --modulepath=/home/michael/provision/modules
  /home/michael/provision/manifests/default.pp
```

Summary

In this chapter we had a whirlwind tour of Puppet, and explored the various ways we could use it to provision a server for our projects. This included:

- Installing software with Puppet
- Managing `cron` jobs with Puppet
- Managing users and groups with Puppet
- Running commands with Puppet
- How modules, classes, and stages work
- How to use Puppet to provision a machine

In the next chapter we will look at Chef, another provisioning tool which has support built into Vagrant.

4
Provisioning with Chef

Chef, as with Puppet, is another provisioning tool which makes it easy for us to take a base operating system install and turn it into a full-fledged server suited to the needs of our project.

In this chapter, we are going to quickly look through the basics of Chef. We won't look at it within a Vagrant context just yet; we will simply look at how Chef works, and how we can use it. In *Chapter 5*, *Provisioning with Vagrant using Puppet and Chef*, we will look at how to connect what we will learn in this chapter with Vagrant itself. In this chapter we will learn:

- How Chef works
- The basics behind Chef cookbooks and recipes
- How to use Chef to perform the following tasks:
    - Install a software
    - Manage files and folders within the filesystem
    - Manage `cron` jobs
    - Run commands
    - Manage users and groups
- How to use third party Chef cookbooks and recipes
- How to manually run Chef to provision a machine

Chef itself is a large topic, and subject of several books. For a more detailed look at Chef, Packt has some titles dedicated to provisioning with Chef:

- *Chef Infrastructure Automation Cookbook*, `http://www.packtpub.com/chef-infrastructure-automation-cookbook/book`
- *Instant Chef Starter*, `http://www.packtpub.com/chef-starter/book`

Knowing about Chef

Chef is a provisioning tool which we can use to set up a server for use for a project. The configuration which determines how the server needs to be setup can be stored within our Vagrant project and can be shared with teammates through version control, ensuring everyone gets an up-to-date copy of the required development environment.

Information about how a server should be configured, that is. Its software, files, users, and groups, is written into files known as Chef recipes. These recipes are written as Ruby files. Chef takes this information and matches it to providers which are used to execute the configuration on the machine in a compatible way.

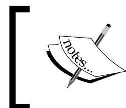
For our purposes we will be using Chef Solo, which is its standalone mode (this is also how Vagrant uses it). Chef Solo means that everything runs from one machine. Chef also has client-server capabilities where you can define the Chef cookbooks and roles for all the servers in your infrastructure on a central host, and it keeps your individual servers at the required level of configuration.

As with Puppet, Chef is also idempotent, which means running Chef on a machine multiple times has the same effect as running it only once.

Creating cookbooks and recipes with Chef

Chef instructions are recipes, which are bundled together in cookbooks. A cookbook contains at least one recipe, which performs some actions. Cookbooks can contain multiple recipes and other resources such as template and files.

At its most basic level, a cookbook is a folder (named as the name of the cookbook) containing at least a recipes folder which contains at least a default recipe file, `default.rb`. Files are typically stored in a files folder within the cookbook folder, and template files within the templates folder.



While both, Puppet and Chef use Ruby, Puppet is a Domain Specific Language which makes it look and feel like its own language, whereas Chef is structured more like Ruby itself.

Resources – what Chef can do

Chef uses resources to define the actions and operations, which can be performed against the system. Resources are mapped to a Chef code, which varies depending on the platform/operating system being used. For example, on an Ubuntu machine the `package` resource is mapped to `apt-get`. Some of these system specific instructions can also be accessed directly via their own resources, `apt_package`, for example, is used to manage packages on Ubuntu and Debian based systems, whereas by using the `package` resource Chef will work out which package manager to use based on the operating system.

Resource types available include:

- `cron`: To manage `cron` jobs on Linux and Unix based systems
- `execute`: To run commands at the terminal/command prompt
- `file`: To manage and manipulate files and folders on the filesystem
- `group`: To manage user groups
- `package`: To install software
- `service`: To manage running services on the machine
- `template`: To manage file contents with an embedded Ruby template
- `user`: To manage user accounts on the machine

Each resource can be configured with different attributes, as we will discuss in this chapter. A full list of the resource types is available from the Opscode website: `http://docs.opscode.com/resource.html`.

Installing software

Let's say we want to install Apache on our server. There are three typical steps involved in this process:

1. Update our package manager
2. Install the Apache package
3. Run the Apache service

Because the first step is different depending on the operating system we are running, we might want to move this out of Chef at a later stage; however we will use it within Chef for the time being. Any operating specific commands (such as this) are written for Ubuntu, which is the operating system we are using with Vagrant.

Updating our package manager

In order to update our package manager, we need to run a command on the server. This can be achieved by using the `execute` resource within Chef, as follows:

```
execute "apt-get update" do
  ignore_failure true
end
```

This instructs Chef to run the `apt-get update` command. As the name of the resource (the part provided in quotes after the name of the resource) is the command we want to run, this will be executed. If we used a friendly name instead, then we would need to provide a command attribute as follows:

```
execute "update-package-manager" do
  command "apt-get update"
  ignore_failure true
end
```

By default, `execute` resources have a timeout of 3600 seconds, however, this can be overridden by providing a timeout attribute to the resource and a time value, for example:

```
execute "apt-get update" do
  ignore_failure true
  timeout 6000
end
```

Installing the Apache package

We can use the `package` resource to ensure that Apache is installed, and if it isn't, it will be installed, as follows:

```
package "apache2" do
  action :install
end
```

Here, we have told Chef to ensure that the `Apache2` package is installed. Provided we have included the recipe or cookbook containing the `apt-get update` command before we include the preceding code, then our package manager will be up-to-date.

Running the Apache service

Finally, make sure that Apache uses the `service` resource. While Apache will automatically run when we install it, we may connect to our new server and alter settings or services by mistake. If this happens, we can simply re-run the provisioner; as Apache will already be installed it won't re-install it, but the `service` resource will force Chef to enable the Apache service (so it automatically starts on system boot) and start the service when the command is run, as follows:

```
service "apache2" do
  supports :status => true, :restart => true, :reload => true
    action [ :enable, :start ]
end
```

Understanding file management

We can use `cookbook_file`, `directory`, `link`, and `template` resources within Chef to manage files and folders within the filesystem. Let's look at some examples which allow us to:

- Copy files
- Create symlinks
- Create folders
- Create multiple folders in one go

Copying a file

One common file operation we might want to perform would be to take a configuration file from our project and copy it into our virtual machine. One particular use case would be an Apache configuration file; we might want to define some virtual hosts and other settings in a file which we can share with our colleagues.

While this works well and can get us up and running quickly, there are modules out there which allow us to configure Apache and other software directly from Chef. This typically works by the module storing a template file (a copy of the configuration file with placeholders in it) and then inserting data we define within Chef into the template and copying the file onto the machine. However, for the sake of this introductory chapter we will just copy a file across.

The `cookbook_file` resource allows us to copy a file from a Chef cookbook onto the machine, as follows:

```
cookbook_file "/etc/apache2/sites-available/default" do
  backup false
  action :create_if_missing
end
```

Because we have omitted the source and path attributes, Chef makes some assumptions. It takes the basename (in effect the last element) of the name and uses this as the source (the basename of `/etc/apache2/sites-available/default` being default) and uses the name as the path (destination). The source file should be located in the files folder within the cookbook's own folder.

As this is an Apache configuration file, it is worth ensuring Apache is already installed, otherwise Apache would override this when it installs the first time, and wouldn't make the process idempotent. We can do this by notifying the `Apache2` service, for example:

```
cookbook_file "/etc/apache2/sites-available/default" do
  backup false
  action :create_if_missing
  notifies :restart, "service[apache2]", :delayed

end
```

The **delayed** option means all of these restart requests will be queued up and executed at the end of Chef's execution; the opposite of this being **immediately**.

Creating a symlink

The link resource allows us to create symbolic links to the existing files and folders on the filesystem. If, for instance, we wanted to map a public folder within our web servers `root` directory to a folder within our Vagrant shared folder, we can do this as follows:

```
link "/var/www/public" do
  to "/vagrant/src/public"
end
```

Creating folders

We can use the `directory` resource to create folders; this is particularly useful for scenarios such as folders to hold files (avatars, attachments, and so on) uploaded by users of a web application:

```
directory "/var/www/uploads" do
  owner "root"
  group "root"
  mode 00777
  action :create
end
```

We can use the `mode` parameter to set the permissions of the folder and the `owner` and `group` parameters to set the user and groups who own the directory (these also can be used for files we create and manage too). Finally, the `:create` action is used to ensure the folder is created.

Creating multiple folders in a single process with looping

In many web projects, we may need to create a number of folders within our servers or our Vagrant virtual machines. In particular, we may want to create a number of cache folders for different parts of our application, or we may want to create some upload folders.

In order to do this, we can simply create an array containing all of the folders we want to create. We can then use the directory resource type and loop through a list of directory names:

```
%w{dir1 dir2 dir3}.each do |dir|
  directory "/tmp/mydirs/#{dir}" do
    mode 00777
    owner "www-data"
    group "www-data"
    action :create
  end
end
```

Managing cron

The `cron` resource type lets us use Chef to manage `cron` jobs which we need to run on the machine. We provide a name, in this case `web_cron`, the command to run, the user to run the command as, and the times at which to run the command, as shown in the following code:

```
cron "web_cron" do
  action :create
  command "/usr/bin/php /vagrant/cron.php"
    user  "root"
  hour "1-4"
    minute "0,30"
end
```

Chef provides us with various different configuration options to define the times at which a `cron` should be run, these include:

- hour: between 0 and 23 inclusive
- minute: between 0 and 59 inclusive
- month: between 1 and 12 inclusive
- day: between 1 and 31 inclusive
- weekday: Sunday (0) – Saturday (6)

If one of these is omitted from the configuration, then Chef runs it for every one of the available options (that is, if we omit month, it will run for January through to December). When defining the dates and times, we can either provide a range, for example, "1-5", or specifics, such as 1,2,10,12. We can also provide an `emailto` property to e-mail the resulting output from the `cron` to an e-mail address of our choosing.

Running commands

The `execute` resource allows us to run commands through the terminal on the machine we are provisioning. One caveat with the `exec` command is that if you re-provision with Chef it will re-run the command, which depending on the command could be damaging. What we can do with the `execute` resource is set the `creates` parameter. The `creates` parameter tells Chef of a file that will be created when the command is run, if Chef finds that file, it knows that it has already been run, and won't run it again.

Take, for example, the following configuration, this would use the PHP composer tool to download dependencies. The command itself creates a file called `composer.lock` (we could, of course, use the `exec` command itself to create a file manually, perhaps using the `touch` command). Because of the lock file that is created, we can use the `creates` parameter to prevent the command from being executed multiple times when a lock file is found:

```
execute "compose" do
  command "/bin/rm -rfv /var/www/repo/vendor/* && /bin/rm -f
    /var/www/repo/composer.lock && /usr/bin/curl -s
      http://getcomposer.org/installer | /usr/bin/php && cd
        /var/www/repo && /usr/bin/php /var/www/repo/composer
          .phar install"
   creates "/var/www/repo/composer.lock"
  timeout 6000
end
```

Managing users and groups

The `user` and `group` resource types let us create and manage users and groups. There are differences between different operating systems as to what Chef can do with the users and groups and how this works.

Creating groups

The simplest way to create a group is simply to set the `action` to `:create`, as follows:

```
group "wheel" do
  action :create
end
```

Creating users

To create a user, the basic information we should provide is:

- The username
- The fact we want to create the user
- The group (`gid`) the user should be part of
- The shell to use for the user
- The home directory for the user
- If we want Chef to manage the home directory for the user; in this case it will create the folder for us if it does not exist
- The password for the user

The code which will then create our user is as follows:

```
user "developer" do
  action :create
  gid "wheel"
  shell "/bin/bash"
  home "/home/developer"
  supports {:manage_home => true}
  password "passwordtest"
end
```

Updating the sudoers file

It's all well and good being able to create users and groups on our machine, however, one thing that we can't do by using the `user` and `group` resource types is define a user or group as having elevated privileges, unless we make the user a part of the `root` group.

We can use an `exec` command through the `execute` resource to push some text to the end of our `suoders` file; the text we need to push simply tells the file that we want to give the `wheel` group sudo privileges. The command we would need to execute is:

```
/bin/echo \"%wheel  ALL=(ALL) ALL\" >> /etc/sudoers
```

Knowing common resource functionalities

There is also a set of common functionality available to all resources. This common functionality includes:

- The ability to do nothing with the `:nothing` action
- Shared attributes available to all resources: `ignore_failure`, `provider`, `retries`, `retry_delay`, and `supports`
- The `not_if` and `only_if` conditions to ensure actions only run when certain conditions are met; these are commands which are run and depending on their return value, recipes, and resources can be ignored.
- Notifications to instruct other resources that another action has completed, or for a resource to take action if another resource changes (subscribes)

Using Chef cookbooks

There are many existing, well written, re-usable Chef cookbooks freely available to use. The Opscode community site contains a collection of these, `http://community.opscode.com/cookbooks`.

It is always worth looking to see if there is an existing cookbook which solves your problem before writing your own.

Using Chef to provision servers

We are going to look at how to use Chef with Vagrant in *Chapter 5*, *Provisioning with Vagrant using Puppet and Chef*, however, Chef can also be run in its own right. Provided Chef is installed (it will be on most Vagrant base boxes, but if you want to run it on another machine, it might not be, so install it first), you can use the `chef-solo` command, passing with it the location of a configuration file to use and a JSON file which contains attributes we wish to use (this should include the rub list, which is the list of recipes and cookbooks we wish to use) as follows:

```
chef-solo -c /home/michael/chefconfig.rb -j
  /home/michael/attributes.json
```

For more information refer to:

- Chef Solo configuration: `http://docs.opscode.com/config_rb_solo.html`
- Apply recipes to run lists: `http://docs.opscode.com/essentials_cookbook_recipes.html#apply-to-run-lists`
- Anatomy of a Chef run: `http://docs.opscode.com/essentials_nodes_chef_run.html`
- Chef tutorial: `http://jonathanotto.com/blog/chef-tutorial-in-minutes.html`

Summary

In this chapter we had a whirlwind tour of Chef, and explored the various ways we could use it to provision a server for our projects. This included:

- Installing software with Chef
- Managing `cron` jobs with Chef
- Managing users and groups with Chef
- Running commands with Chef
- How recipes and cookbooks work
- How to use Chef to provision a machine

In the next chapter we will look at how to use both Chef and Puppet to provision a machine within the context of Vagrant.

5
Provisioning with Vagrant using Puppet and Chef

In *Chapter 3*, *Provisioning with Puppet*, and *Chapter 4*, *Provisioning with Chef*, we had an introduction to both, Puppet and Chef, which are provisioning tools with support built into Vagrant. We, however, looked at how the tools worked generally; we didn't look at how to use them with Vagrant.

In this chapter you will learn:

- Using Puppet within Vagrant
- Using Chef within Vagrant
- About the other built-in provisioners:
    - Re-capping how to provision with the built-in SSH provisioner
    - Using the **Ansible** provisioner
- Working with multiple provisioners
- How we can override the provisioning tools through the command line

Provisioning within Vagrant

Vagrant relies on base boxes for the guest virtual machines; these are specifically pre-configured VM images, which have certain software packages pre-installed and pre-configured. Puppet and Chef are two such software packages that are pre-installed. Vagrant has its own interface through to these packages from the host machine. This means we can provide some configuration in our Vagrant file and Vagrant will pass this information to the relevant provisioners on the guest VM.

Provisioning with Puppet on Vagrant

Vagrant supports two methods of using Puppet:

- Puppet in a standalone mode, by using the Puppet `apply` command on the VM
- Puppet in client/server mode, whereby the VM (by using the Puppet agent) will be configured from a central server

Let's look at how to configure Vagrant with Puppet using these two different methods.

Using Puppet in a standalone mode

Puppet standalone is the simplest way to use Puppet with Vagrant, we simply tell Vagrant where we have put our Puppet manifests and modules, and what manifest should be run. The smallest amount of configuration we need within our Vagrant file in order to use Puppet is this:

```
config.vm.provision :puppet
```

This should go within the `Vagrant.configure("2") do |config| … end` block of code within the Vagrant file.

Along with this configuration we will need a Puppet manifest called `default.pp` in the `manifests` folder of our project root. Vagrant will then use this to provision the machine.

This will instruct Vagrant to run the Puppet provisioner either when the machine boots up or if we run the Vagrant `provision` command. The default Vagrant Puppet setup will make the following assumptions, unless we override the settings:

- Manifests will be located in the `manifests` folder
- Modules will also be located in the `manifests` folder (we may want to point these elsewhere, especially if we are using third party modules, to keep them separate)
- The manifest file to use will be `default.pp` (and will obviously be within the `manifests` folder; it can be useful to override this if we are using Puppet modules and manifests within the same project for multiple environments, we may have a manifest for our Vagrant VM, one for our production environment and one for a user acceptance testing platform for example)

We can modify these options by provisioning configuration options as opposed to just telling Vagrant to provision with Puppet. When creating projects which are managed by Vagrant, I like to put all my provision related files structured within the provision folder. In order to override these, within the Puppet configuration for Vagrant, we can then specify options for the location of Puppet manifests (`puppet.manifests_path`), the name of the Puppet manifest to apply (`puppet.manifests_file`), and the location of any Puppet modules which we may reference within our Puppet manifest (`puppet.module_path`). The following customizes these options:

```
config.vm.provision :puppet do |puppet|
    puppet.manifests_path = "provision/manifests"
    puppet.manifest_file  = "default.pp"
    puppet.module_path = "provision/modules"
end
```

It is important for us to have the ability to at least change the manifest file, as Vagrant also supports a multi VM environment, where a single project can have a number of virtual machines (for example, a web server and a database server). With this setup, we would need to tell Vagrant which manifest file to use for each of the machines, so that the web server can be properly configured as a web server and the database server as a database server.

Puppet provisioning in action

Our knowledge of creating Puppet modules and manifests from *Chapter 3*, *Provisioning with Puppet*, we can now point our Vagrant configuration at these files and see it in action. If we run a Vagrant file up on a project which is suitably configured, we will see the output of Puppet applying its settings to our VM in the terminal window, as shown in the following screenshot:

```
notice: /Stage[veryfirst]/Dns/Exec[updatedns]/returns: executed successfully
notice: /Stage[veryfirst]/Dns/Exec[preparenetworking]/returns: executed successfully
notice: /Stage[ppa]/Misc/Exec[apt-get update]/returns: executed successfully
notice: /Stage[main]/Sudo/Exec[/bin/echo "%wheel  ALL=(ALL) ALL" >> /etc/sudoers]/returns: executed successfu
notice: /Stage[main]/Php/Package[php5-mhash]/ensure: ensure changed 'purged' to 'present'
notice: /Stage[main]/Mysql/Package[libmysqlclient15-dev]/ensure: ensure changed 'purged' to 'present'
notice: /Stage[main]/Mysql/Exec[apache2ctl graceful]/returns: executed successfully
notice: /Stage[main]/Php/Exec[testing-and-qa]/returns: executed successfully
notice: /Stage[main]/Mail/Exec[autostartmail]/returns: executed successfully
notice: /Stage[last]/Modrewrite/Exec[enabledmodrewrite]/returns: executed successfully
notice: /Stage[last]/Modrewrite/Exec[reload-apache2-again]/returns: executed successfully
notice: Finished catalog run in 382.39 seconds
Michaels-MacBook-Pro:template michael$
```

The console output highlights details of each Puppet instruction that is run, including:

- The stage the instruction is within (this is the Puppet stage, as we discussed in *Chapter 3*, *Provisioning with Puppet*, which allows us to group classes together to control the ordering of certain actions)
- The module
- The resource type
- The resource name
- If the instruction was executed successfully

Using Puppet in client/server mode

As discussed earlier, we can also run Puppet within our Vagrant environment in client/server mode, by using the Puppet agent on the virtual machine. The configuration required for this is minimal, we simply tell Vagrant the address of the Puppet server we are using, and the name of our node (the virtual machine we are setting up):

```
config.vm.provision :puppet_server do |puppet|
  puppet.puppet_server = "puppet.internal.michaelpeacock.co.uk"
  puppet.puppet_node = "vm.internal.michaelpeacock.co.uk"
end
```

The node name is the reference for the machine within the Puppet server so the Puppet server knows how our VM should be configured. The node name is also used to generate an SSL certificate so that the VM can authenticate with the Puppet server (more detail on this is available on the Puppet website, `puppetlabs.com`, and the Puppet blog available at, `puppetlabs.com/blog/deploying-puppet-in-client-server-standalone-and-massively-scaled-environments/`).

Provisioning with Chef on Vagrant

Vagrant also supports two methods of using Chef:

- Chef solo
- Chef in client/server mode with Chef client

Let's look at how to configure Vagrant with Chef using these two different methods.

Using Chef solo

Chef solo is the Chef equivalent of Puppet standalone.

The simplest way to use this within our project is simply to provide a Chef run list to Vagrant, this tells Vagrant which cookbooks should be applied. The following is an example of telling Vagrant to use the PHP cookbook:

```
config.vm.provision :chef_solo do |chef|
  chef.add_recipe "php"
end
```

This takes the PHP cookbook from the default cookbooks folder and applies it to the virtual machine.

As with Puppet, Vagrant makes some assumptions by default, these are:

- Cookbooks are stored in the cookbooks folder within the project root.

  The `chef.cookbooks_path` setting allows us to override the cookbooks folder location; we can either provide a single path or we can provide an array of paths (wrapped in square brackets, separated with commas) if we want Vagrant and Chef to look in a range of folders for our cookbooks. The following code would go into our Vagrant file to tell Vagrant to override the cookbooks folder with `provision/cookbooks`:

  ```
  config.vm.provision :chef_solo do |chef|
    chef.cookbooks_path = "provision/cookbooks"
  end
  ```

- We can also use Chef Roles by providing:
  - The location of the roles folder
  - The roles we wish to add to the VM

  More information on Chef Roles can be found on the Opscode website `http://docs.opscode.com/essentials_roles.html`.

  The following code in our Vagrant file would set up our project to use Chef Roles:

  ```
  config.vm.provision :chef_solo do |chef|
    chef.roles_path = "provision/roles"
    chef.add_role("web")
  end
  ```

Using Chef in a client/server mode

Like Puppet, Chef also has a client/server method for provisioning machines, by using Chef Client on the VM. To use Chef Client we need to tell Vagrant where the Chef Server is located (through the `chef.chef_server_url` setting) and provide the authorization key which will be used to authenticate the VM with the Server (through the `chef.validation_key_path` setting).

The following code in our Vagrant file will instruct Vagrant to provision from a Chef Server:

```
config.vm.provision :chef_client do |chef|
  chef.chef_server_url = "http://chef.internal.michaelpeacock.
    co.uk:4000/" chef.validation_key_path = "key.pem"
end
```

We can also override the run list that the Chef Server provides for our VM by manually adding roles and recipes to this configuration. If we have specified different environments on our Chef Server, we can specify which environment we want to use with the `chef.environment` configuration.

Vagrant VMs which use Chef Server will have corresponding node and client entries on the Chef Server named with the hostname of the VM. If we destroy the VM and recreate it, Chef will generate an error because the client and node entries are already present on the server. We need to remove these from the Chef Server when we destroy a VM. This can be done by using the knife tool from chef: `knife node delete precise64 && knife client delete precise63`.

Other built-in provisioners

In addition to the Puppet and Chef provisioning options within Vagrant, there are two other methods:

- SSH: Simply invoking commands via the terminal of the VM automatically through Vagrant
- Ansible: A tool similar in nature to Puppet and Chef, which is configured through a series of YAML files to define how a system should be provisioned

Provisioning with SSH – a recap

As we discussed in *Chapter 2, Managing Vagrant Boxes and Projects*, we can instruct Vagrant to run a series of SSH commands on the VM. This can be used to provision the server.

There are two ways to use SSH provisioning:

- Path: a file to execute
- Inline: providing specific commands to run

Both of these are shown as follows:

```
config.vm.provision :shell, :path => "provision/setup.sh"

config.vm.provision :shell, :inline => "apt-get install apache2"
```

Ansible playbooks

Detailed information on Ansible can be found on the project's website (`http://www.ansibleworks.com/`). In order to use Ansible within a Vagrant project, we need to tell Vagrant where the playbook and inventory files are:

```
config.vm.provision :ansible do |ansible|
  ansible.playbook = "provision/playbook.yml"
  ansible.inventory_file = "provision/hosts"
end
```

The inventory file contains a list of environment names and IP addresses; we use this to restrict which commands Ansible runs on specific environments.

Using multiple provisioners on a single project

We can use multiple provisioners within a single project if we wish; we simply need to put them in the order we wish for them to be executed within our Vagrant file. The following would first run an SSH command before provisioning with Puppet:

```
Vagrant.configure("2") do |config|
  Config.vm.box = "precise64"

  config.vm.provision :shell, :inline => "apt-get update"
```

```
    config.vm.provision :puppet do |puppet|
         puppet.manifests_path = "provision/manifests"
         puppet.manifest_file  = "default.pp"
         puppet.module_path = "provision/modules"
    end

  end
```

Using multiple provisioners can be useful, especially if one is more suited at specific tasks than another, or if we require some prerequisites. For example, when using Puppet and Chef in the client/server mode, they need to have an SSH key to communicate with the server. We could use a shell provisioner to instruct the VM to download the keys we have prepared from a secure location before the Puppet or Chef provisioners kick in.

Overriding provisioning via the command line

There may be instances where we want to restrict or enforce the execution of provisioning or even a specific provisioner within a project. The following commands are all executed from the host machine:

- We can cancel a running provision by pressing *CMD* + *C* at the terminal
- We can instruct Vagrant to re-run provisioning on a VM by using the `vagrant provision` command
- We can also add `no-provision` to the `up` and `reload` commands to instruct Vagrant to not run the provisioning tools when performing the up and reload actions
- We can also provision with just a specific provisioner should we wish, for example, if we wanted to instruct Vagrant to just run Puppet in a standalone mode (in a project which has multiple provisioners configured) we would run vagrant provision `provision-with` puppet

Summary

In this chapter we have learned how we can apply our knowledge of Puppet and Chef from *Chapter 3*, *Provisioning with Puppet*, and *Chapter 4*, *Provisioning with Chef*, and configure Vagrant to use these tools to provision our virtual machines.
This has included:

- Using Puppet in a standalone mode, which uses the Puppet `apply` command to apply locally stored manifests and modules onto the machine
- Using Puppet in the client/server mode, which uses the Puppet agent to retrieve configuration from a central server to provision the machine
- Using Chef solo which applies locally stored cookbooks and recipes to the machine
- Using Chef in the client/server mode, which uses the Chef client to retrieve the configuration from a central server to provision the machine
- Other provisions including SSH provisioning and the Ansible provisioner
- Running multiple provisioners within a single project
- Overriding provisioning at the command line and re-running the provisioning tools with vagrant provision

Now, we have fully mastered how to set up, use, and manage Vagrant along with the provisioning tools to work on a single machine project. In *Chapter 6*, *Working with Multiple Machines*, we will look at how to use Vagrant and our knowledge of provisioners to manage a multi-machine project, with provisioners configuring different machines for different purposes for use within the project for example, a web server and a database server.

6
Working with Multiple Machines

So far we have used Vagrant to build a development environment contained within a virtual machine; one of the key aspects being that this virtual machine mimics our production environment. It gives us the flexibility of being able to encapsulate the development environment for different projects so that we can easily switch from one to another without having to modify the software on our own machines.

In many cases, the features we have learned so far are enough. However, web projects are more and more complex, with developers continually improving, having to deal with multiple machines in their architecture to help with scalability and stability. While it can be said that scalability and stability issues won't impact on our development environment (as we won't have huge amounts of traffic coming to our development environment, unless we load test it), we want to ensure the coupling between servers within our code (such as application code connecting to a remote database) works in our development environment before we put the project online.

Thankfully, Vagrant has support for running multiple virtual machines at the same time within the same project. We can use this to test multi-machine architectures and distributed systems on our own local machine before we share our changes with colleagues in a staging environment and before the project goes live. Replicating a multi-machine environment in development greatly helps us improve the reliability of our projects and instills confidence in the work that we do.

In this chapter you will learn:

- How to run multiple virtual machines within a single Vagrant project
- How to provide different distinct configurations to these virtual machines including:
    - Names
    - IP addresses on a private network so they can communicate with one another
    - Base boxes
    - Provisioning
    - Shared folders
- How to connect to the different virtual machines over SSH without having to know or remember their IP addresses

Using multiple machines with Vagrant

In order to use multiple virtual machines within our project, we need to tell Vagrant about them, and we need to provide additional configuration for the individual virtual machines.

Defining multiple virtual machines

Within the standard Vagrant project configuration file, we can tell Vagrant that we wish to assign a name to a virtual machine being managed by the project. Within this subconfiguration, we provide the information Vagrant needs which are specific to that VM.

The syntax for the subconfiguration is:

```
config.vm.define :name_of_the_vm do |name_of_the_vm|
    #configuration specific to the virtual machine
end
```

Applied to a project which requires two virtual machines, named `server1` and `server2`, both running the `precise64` box:

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

Vagrant.configure("2") do |config|

  config.vm.define :server1 do |server1|
    server1.vm.box = "precise64"
```

```
    end

    config.vm.define :server2 do |server2|
      server2.vm.box = "precise64"
    end

  end
```

Connecting to multiple virtual machines over SSH

When our multiple machines boot up in our multi-machine project, Vagrant automatically maps different ports from our host machine to the SSH ports on the various guest machines.

Let's look at the console output when booting a Vagrant project with two virtual machines within it:

```
Michaels-MacBook-Pro:mutlimachine michael$ vagrant up
Bringing machine 'webserver' up with 'virtualbox' provider...
[webserver] Importing base box 'precise64'...
[webserver] Matching MAC address for NAT networking...
[webserver] Setting the name of the VM...
[webserver] Clearing any previously set forwarded ports...
[webserver] Creating shared folders metadata...
[webserver] Clearing any previously set network interfaces...
[webserver] Preparing network interfaces based on configuration...
[webserver] Forwarding ports...
[webserver] -- 22 => 2222 (adapter 1)
[webserver] Booting VM...
[webserver] Waiting for VM to boot. This can take a few minutes.
[webserver] VM booted and ready for use!
[webserver] Configuring and enabling network interfaces...
[webserver] Mounting shared folders...
[webserver] -- /vagrant
Bringing machine 'database' up with 'virtualbox' provider...
[database] Importing base box 'precise64'...
[database] Matching MAC address for NAT networking...
[database] Setting the name of the VM...
[database] Clearing any previously set forwarded ports...
[database] Fixed port collision for 22 => 2222. Now on port 2200.
[database] Creating shared folders metadata...
[database] Clearing any previously set network interfaces...
[database] Preparing network interfaces based on configuration...
[database] Forwarding ports...
[database] -- 22 => 2200 (adapter 1)
[database] Booting VM...
[database] Waiting for VM to boot. This can take a few minutes.
[database] VM booted and ready for use!
[database] Configuring and enabling network interfaces...
[database] Mounting shared folders...
[database] -- /vagrant
```

As shown in the preceding screenshot, Vagrant maps the SSH port on the virtual machine designated **'webserver'** to **port 2222** on the host machine, and the SSH port of the machine designated **'database'** to the **port 2200**.

This gives us the opportunity of simply using the standard SSH command from a terminal (or the likes of Putty on a Windows machine) to connect to `localhost` with the port number that Vagrant assigns to each machine.

To connect to the machine which is mapped to port 2200 we simply run the following command:

```
ssh vagrant@localhost -p2200
```

`-p2200` tells the command to use a non-standard port, and specifies the port we wish to use, in this case 2200. As you can see, this then lets us into the appropriate machine, as follows:

Alternatively, we can use the `vagrant ssh` command to connect to the virtual machines. The difference being that in a multi virtual machine environment, we must also provide the name of the virtual machine. For example, `vagrant ssh database`. This is the most common usage of connecting to a machine, rather than directly connecting to the virtual machine via its IP address.

Networking multiple virtual machines

In a single virtual machine project, the IP address of the virtual machine isn't that important. In a multi virtual machine project however, it is more likely that we want the two machines to communicate with one another directly; in order to do that, we need to be aware of their IP addresses. As we want to have our Vagrant projects distributed to our team members, and some of these team members may be within the same office, we need to:

- Predefine the IP address so that any of our projects code which needs to communicate with the other virtual machine can do so without the other team members needing to change configurations
- Ensure that the virtual machines are running on a private network only attached to the machine of the user running it; this will prevent IP address conflicts within the network

In order to do this, we simply use the networking options which we learned about in *Chapter 2, Managing Vagrant Boxes and Projects*. Because we want the virtual machines running in a private network, it makes sense to use a range of private IP addresses which are different to your own network. For example, my network range is `192.168.1.xxx`, so I will use the range `10.11.1.xxx` for my virtual machine network, as shown in the following code:

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

Vagrant.configure("2") do |config|

  config.vm.define :server1 do |server1|
    server1.vm.box = "precise64"
    server1.vm.network :private_network, ip: "10.11.1.100"
  end

  config.vm.define :server2 do |server2|
    server2.vm.box = "precise64"
    server2.vm.network :private_network, ip: "10.11.1.101"
  end

end
```

Let's test this out and test that we can connect from one machine to the other:

1. Power up the project (`vagrant up`)
2. Connect to `server1` (`vagrant ssh server1`)
3. Ping `server2` from `server1` (`ping 10.11.1.101`)

The output shows that we are able to communicate over the network from `server1` to `server2` as follows:

```
Michaels-MacBook-Pro:mutlimachine michael$ vagrant ssh server1
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-23-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Welcome to your Vagrant-built virtual machine.
Last login: Fri Sep 14 06:23:18 2012 from 10.0.2.2
vagrant@precise64:~$ ping 10.11.1.101
PING 10.11.1.101 (10.11.1.101) 56(84) bytes of data.
64 bytes from 10.11.1.101: icmp_req=1 ttl=64 time=0.779 ms
64 bytes from 10.11.1.101: icmp_req=2 ttl=64 time=0.378 ms
64 bytes from 10.11.1.101: icmp_req=3 ttl=64 time=0.402 ms
64 bytes from 10.11.1.101: icmp_req=4 ttl=64 time=0.370 ms
64 bytes from 10.11.1.101: icmp_req=5 ttl=64 time=0.517 ms
64 bytes from 10.11.1.101: icmp_req=6 ttl=64 time=0.351 ms
^C
--- 10.11.1.101 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5000ms
rtt min/avg/max/mdev = 0.351/0.466/0.779/0.150 ms
vagrant@precise64:~$
```

Provisioning the machines separately

As the virtual machines in our projects are going to be used for different purposes, we need to use different provisioning for the machines so they both have the software and configurations needed to do their job.

We take the provisioning code, which we have learned in *Chapter 3*, *Provisioning with Puppet*, and *Chapter 4*, *Provisioning with Chef*, and we place the relevant code within the virtual machines subconfiguration. There are some key changes which we need to make:

- The opening line of the provision code must reference the server name of the virtual machine it relates to
- For Puppet, we should use a different manifest file for the two virtual machines
- For Chef, we would apply different roles to different machines

The following code provisions both the machines by using Puppet. They both rely on the same set of Puppet modules, the same path pointing to the manifests folder, however they both use different manifests to set up the projects (alternatively, we could configure the machines, identify themselves as nodes to a puppet master to retrieve the appropriate configuration):

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

Vagrant.configure("2") do |config|

  config.vm.define :server1 do |server1|
    server1.vm.box = "precise64"
    server1.vm.network :private_network, ip: "10.11.1.100"
    server1.vm.provision :puppet do |puppet|
      puppet.manifests_path = "provision/manifests"
      puppet.manifest_file  = "server1.pp"
      puppet.module_path = "provision/modules"
    end
  end

  config.vm.define :server2 do |server2|
    server2.vm.box = "precise64"
    server2.vm.network :private_network, ip: "10.11.1.101"
    server2.vm.provision :puppet do |puppet|
      puppet.manifests_path = "provision/manifests"
      puppet.manifest_file  = "server2.pp"
      puppet.module_path = "provision/modules"
    end
  end

end
```

Within the provisions for each machine, we would need to ensure that we allow both machines to communicate with one another. For example, by default a MySQL server won't accept connections from a remote server, so it would be needed to modify (or replace) the configuration file with one which allows it, or we would have to use a Puppet module or Chef cookbook which allowed us to modify this configuration value.

> You should check the documentation for any software you are communicating with over the network to see how it needs to be configured. With MySQL you need to edit the `my.cnf` file and set the bind address to `0.0.0.0`.

Destroying a multi-machine project

If we want to completely remove the virtual machines for our project from our host machine, we can use the `vagrant destroy` command as with normal projects. The difference being that Vagrant will ask us to confirm the removal of each machine within the project, as shown in the following screenshot:

```
Michaels-MacBook-Pro:mutlimachine michael$ vagrant destroy
Are you sure you want to destroy the 'database' VM? [y/N] y
[database] Forcing shutdown of VM...
[database] Destroying VM and associated drives...
Are you sure you want to destroy the 'webserver' VM? [y/N] y
[webserver] Forcing shutdown of VM...
[webserver] Destroying VM and associated drives...
```

Summary

In this chapter, we set up a Vagrant project which uses multiple virtual machines. During the course of this we learned:

- How to create multiple virtual machines within a single project
- How to assign names to the individual machines
- How to connect to the individual machines over SSH by using both the operating systems built in the SSH command and the `vagrant ssh` command
- How to configure the individual virtual machines within the project, providing IP addresses, base boxes, and provisioning options to them

Now we have learned the vast majority of Vagrants functionality and how to use it within different project scenarios. In the next chapter, we will look at how to build our own custom base box to use with our projects, configuring a blank operating system installation into a compatible base image.

7
Creating Your Own Box

So far, we have used Vagrant with the freely available base boxes, and learned about the websites available such as `vagrantbox.es`, which provide a list of unofficial third-party base boxes. When we discussed Vagrant boxes initially, we also learned about how we can export a Vagrant environment into a new base box.

This involves us either finding a base box or customizing an existing base box. In this chapter, we will look at how we can take a Linux installation disk and turn it into a working Vagrant base box, which we can further customize as much as we like.

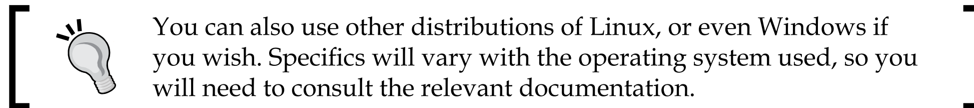
In this chapter, you will learn:

- How to create a new VirtualBox machine, suitably configured for Vagrant
- How to install the VirtualBox Guest Additions
- How to set up the Linux installation to let Vagrant log in
- How to install Puppet
- How to install Chef
- How to clean up the box
- How to export the VM into a base box

Getting started

While working with an older version of Vagrant, the documentation at `http://docs-v1.vagrantup.com/v1/docs/base_boxes.html` may be a useful reference to accompany this chapter.

In order for us to create a new base box, we need to download a copy of the operating system we want to use (we will use 64-bit Ubuntu Server Version 13.04 from `http://www.ubuntu.com/download/server`). We then need to use VirtualBox to create a virtual machine, powered by the operating system we have downloaded. Next, we need to configure the virtual machine for Vagrant. Finally, we need to export the virtual machine into a Vagrant base box.



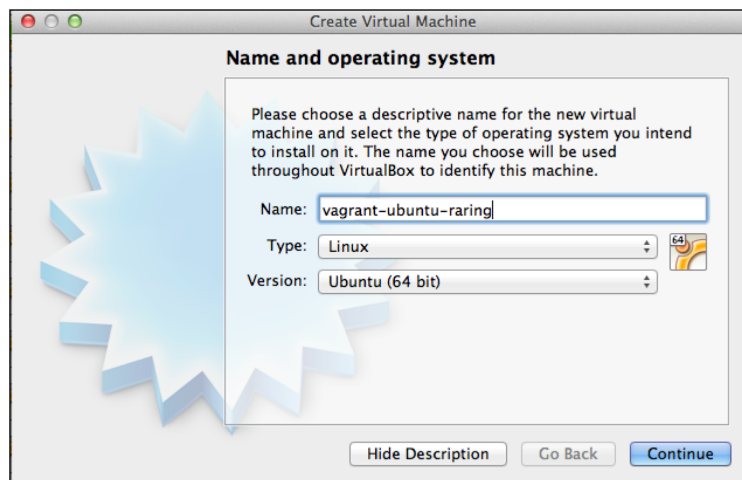
You can also use other distributions of Linux, or even Windows if you wish. Specifics will vary with the operating system used, so you will need to consult the relevant documentation.

Preparing the VirtualBox machine

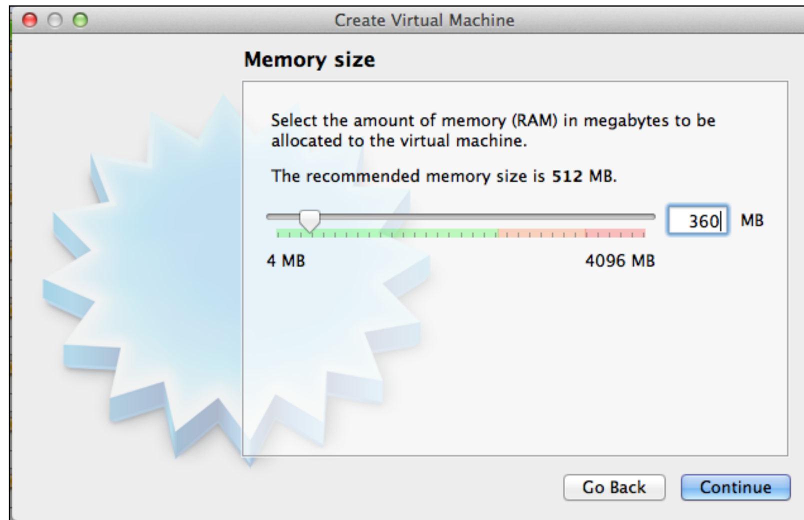
In order to create the virtual machine with VirtualBox, we need to open up the VirtualBox and perform the given steps:

1. Click on the **New** button, at the upper-left corner of the VirtualBox, to start the process:

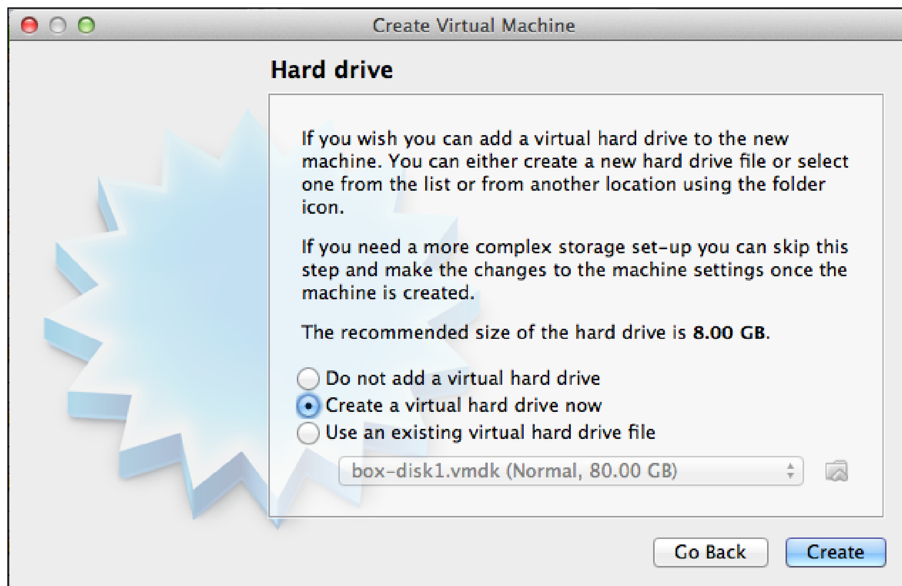
2. Let's name the machine as `vagrant-ubuntu-raring`, this is the format recommended by Vagrant. Select **Linux** in the **Type** drop-down, and **Version** as **Ubuntu (64 bit)**:



3. Vagrant recommends setting a memory allocation of 360 MB. This is typically sufficient for a base install, and users can override this within their `Vagrantfile` if they need more resources:

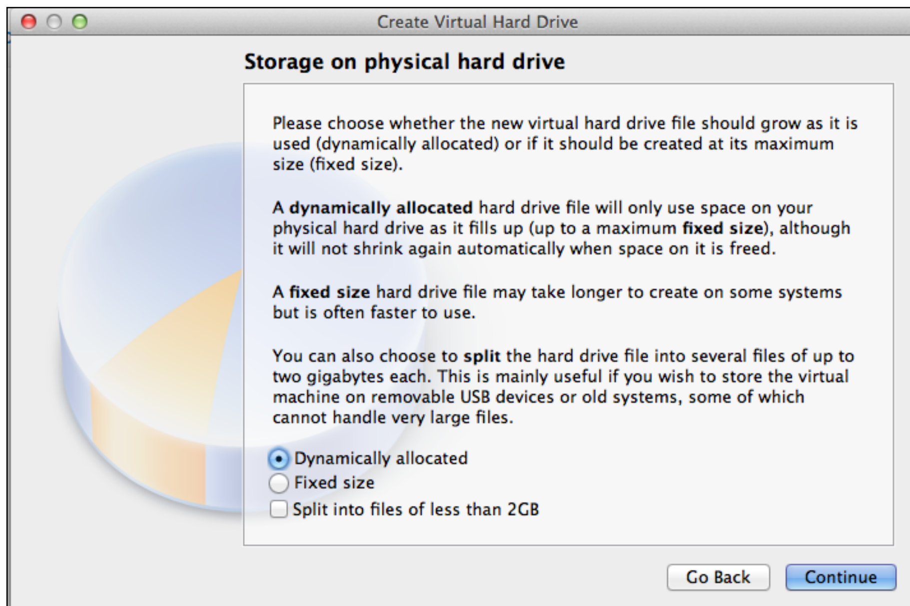


4. We need our virtual machine to have some storage allocation, so let's select **Create a virtual hard drive now**:

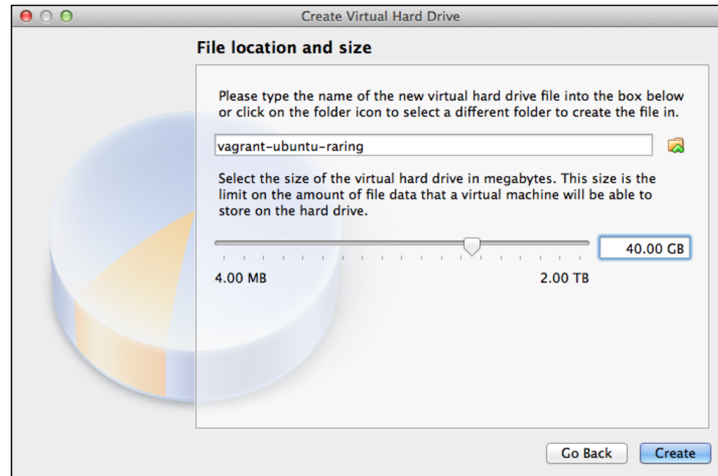


5. We want to select **VMDK (Virtual Machine Disk)** as the disk type:

6. We want to create a drive, which is **Dynamically allocated**:



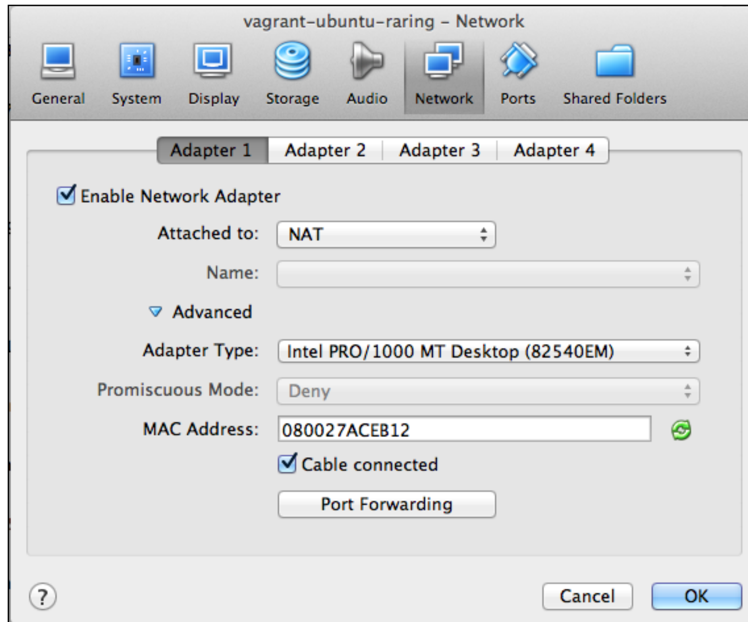
7. Let's give the drive a maximum limit of `40.00 GB`; the Vagrant documentation suggests that this is typically sufficient for many projects:



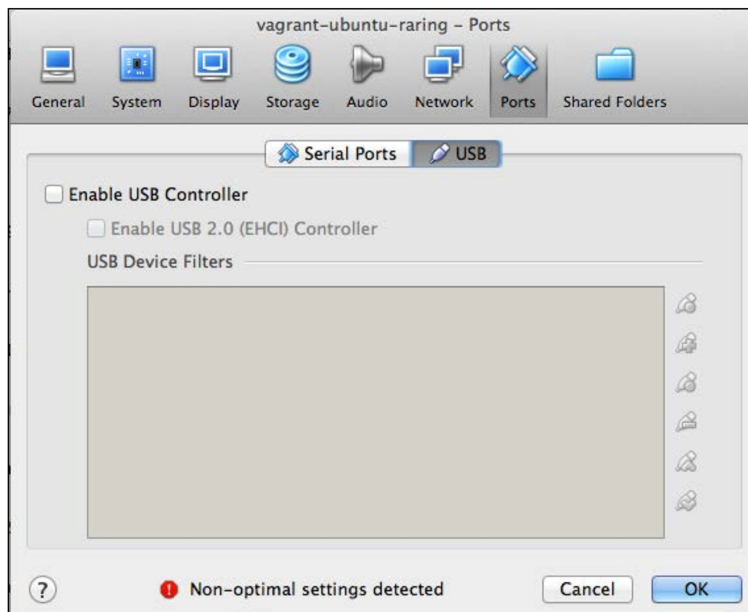
8. Clicking on **Create** will then save the virtual machine within VirtualBox. We need to make some additional configuration changes which were not a part of the creation wizard, so let's click on the VM from the left-hand side of the screen and then click on the **Settings** button:

9. The first additional change is **Audio**; let's turn this off:

10. We need to ensure that the network adapter configured within VirtualBox is enabled and uses **NAT**. Without this, Vagrant won't work:



11. Finally, let's turn off USB support, as this is generally not required:



12. Now we need to switch on the virtual machine. When it powers on, it asks us to select a startup disk, which contains the operating system we wish to install. Clicking on the folder icon on this screen lets us select a custom file, in our case, this will be our `ubuntu-13.04-server-amd64.iso` file.

The virtual machine will then boot from the CD and take us to the installation process. We should follow this process to set up the machine.

There are some specific values for some things which Vagrant expects, so wherever appropriate, we should ensure we set them as such:

- By convention, the operating system's hostname should be of the format `vagrant-operating-system-name`, for example, `vagrant-ubuntu-raring`
- Domain: `vagrantup.com`
- Root password: `vagrant`
- Main account username: `vagrant`
- Main account password: `vagrant`

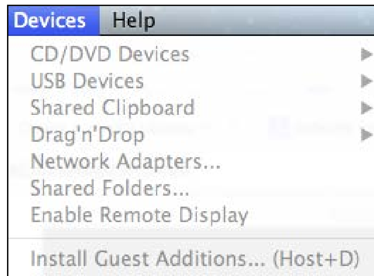
In most other cases, the default options will be fine, as we will configure other aspects later. When prompted as to any packages to install by default, we should select to install `openssh-server`.

VirtualBox Guest Additions

First, let's log in to our new virtual machine within VirtualBox. Once logged in, at the terminal, we should run `apt-get update` to update our package manager.

Vagrant has a set of tools called Guest Additions, which provide some key integration points between the virtual machine and VirtualBox; this includes support for shared folders and networking integration.

To install these tools, once the VM is running, we should click on the **Devices** menu within VirtualBox and click on **Install Guest Additions...**:



This simply boots a virtual CD within the virtual machine; we still need to actually install the Guest Additions. The first step of that is to install a prerequisite, which are the Linux headers:

```
sudo apt-get install linux-headers-$(uname -r) build-essential
```

Next, we want to mount the virtual CD which VirtualBox has loaded up into a folder within the VM:

```
sudo mount /dev/cdrom /media/cdrom
```

Finally, we want to run the installation command:

```
sudo sh /media/cdrom/VBoxLinuxAdditions.run
```

Vagrant authentication

Vagrant communicates with base boxes over SSH. Vagrant itself has a private key, for which we need to install the corresponding public key into the virtual machine. Vagrant expects a specific user with a predefined password to also be within the machine, and the user needs to be configured so that it isn't prompted for the password when attempting to perform actions which require elevated privileges (`sudo`).

Vagrant user and admin group

Provided we created the Vagrant user during the installation process (as per the main account user and password mentioned earlier), and then we need to create an `admin` group and add the Vagrant user to this group.

Firstly, to create the group:

```
Sudo groupadd admin
```

To add the Vagrant user to this group:

```
Sudo usermod -a -G admin vagrant
```

Sudoers file

In order to stop the virtual machine asking for the user's password when running elevated actions, we need to modify the `sudoers` file (this is a file which tells the operating system which users can perform elevated actions and the settings around them. More information can be found at `https://help.ubuntu.com/community/Sudoers`). We need to add to this file a configuration line, which tells the operating system not to prompt for the password. Because the file is very important, and an incorrect configuration would break the operating system, there is a program built into Ubuntu, which won't save if the file is not edited correctly.

First, let's run this program:

```
visudo
```

At the bottom of the file, let's add this line to prevent the operating system prompting for the password:

```
%admin ALL=(ALL) NOPASSWD: ALL
```

Another requirement of Vagrant is that we add the following line near the top of this file:

```
Defaults env_keep="SSH_AUTH_SOCK"
```

We also need to disable `requiretty` in the `sudoers` file by commenting out the appropriate line as:

```
#Default requiretty
```

Insecure public/private key pair

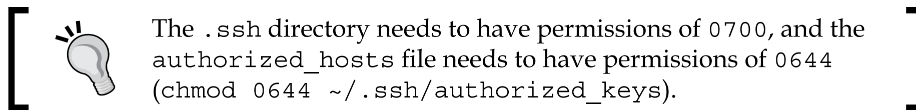
The insecure public and private key pair is publicly available at `https://github.com/mitchellh/vagrant/tree/master/keys/`

We need to copy the contents of the public key and paste it into the `authorized_hosts` file. Provided we are logged in as the Vagrant user, we can run the following command to let us edit this file:

```
nano ~/.ssh/authorized_hosts
```

Alternatively, we can download the file contents and put it straight into the `authorized_hosts` file:

```
wget
  https://raw.github.com/mitchellh/vagrant/master/keys/
    vagrant.pub -o ~/.ssh/authorized_hosts
```



The `.ssh` directory needs to have permissions of `0700`, and the `authorized_hosts` file needs to have permissions of `0644` (`chmod 0644 ~/.ssh/authorized_keys`).

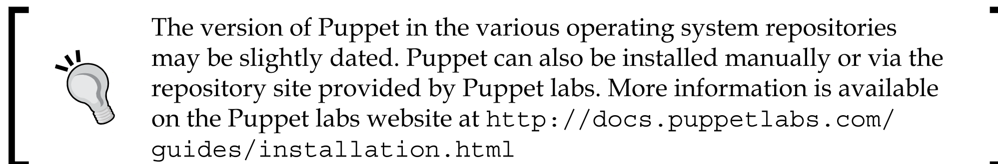
Provisioners

Because Vagrant provides support for provisioners, we need to install these into the virtual machine so that Vagrant can tell them to provision our environments.

Puppet

Puppet is installed using the built-in package manager:

```
sudo apt-get install puppet
```



The version of Puppet in the various operating system repositories may be slightly dated. Puppet can also be installed manually or via the repository site provided by Puppet labs. More information is available on the Puppet labs website at `http://docs.puppetlabs.com/guides/installation.html`

Chef

As per the Chef documentation at `http://wiki.opscode.com/display/chef/Installing+Chef+Client+on+Ubuntu+or+Debian`, it is recommended that we manually install RubyGems and then use this to install Chef. RubyGems has some dependencies, so let's install them first:

```
sudo apt-get install ruby ruby-dev libopenssl-ruby rdoc ri irb
    build-essential wget ssl-cert curl
```

Next, let's install RubyGems:

```
cd /tmp
curl -O http://production.cf.rubygems.org/rubygems/
    rubygems-1.8.10.tgz
tar zxf rubygems-1.8.10.tgz
cd rubygems-1.8.10
sudo ruby setup.rb --no-format-executable
```

Now, let's install Chef:

```
sudo gem install chef --no-ri --no-rdoc
```

Cleanup

Before we package up the virtual machine into a Vagrant base box, let's cleanup some of the files we used. We made use of the `tmp` folder, so let's empty that. We should also clean up our package manager's cache, as this uses additional space when the base box is packaged:

```
rm –rf /tmp/*
sudo apt-get clean
```

Export

Finally, we use Vagrant's `package` subcommand to package up the box:

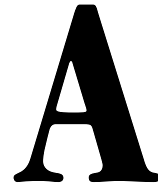
```
vagrant package --base vagrant-ubuntu-raring
```

Full details of the `package` subcommand are available on the Vagrant website: `http://docs.vagrantup.com/v2/cli/package.html`

Summary

In this chapter, we learned how to create from scratch, a base box for our Vagrant projects. This can be used to create base boxes from operating systems, which don't necessarily have boxes available to download.

Now, we know how to create, manage, distribute, and even build development environments from scratch for our projects!



A Sample LAMP Stack

Now that we have a good knowledge of Vagrant, how to use it to manage projects, and how to use the Puppet provisioning tool, let's look at how to use these tools to build a **LAMP** (**Linux, Apache, MySQL, and PHP**) development environment with Vagrant.

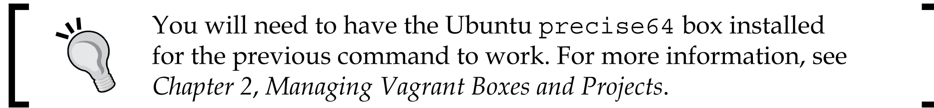
In this chapter you will learn:

- How to update our package manager
- How to create a LAMP-based development environment in Vagrant, including:
  - How to install Apache
  - How to have Apache reload when we install PHP
  - How to install and enable Apache's rewrite module
  - How to customize the Apache configuration file
  - How to install PHP
  - How to install MySQL
  - How to install e-mail sending services

Creating the Vagrant project

First, we want to create a new project, so let's create a new folder called `lamp_stack` and initialize a new `precise64` Vagrant project within it by executing the following commands:

```
mkdir lamp_stack
cd lamp_stack
vagrant init precise64
```



You will need to have the Ubuntu `precise64` box installed for the previous command to work. For more information, see *Chapter 2, Managing Vagrant Boxes and Projects*.

We want to forward port 80 from our guest machine to port 8080 on our host machine to make it easier to access the web project we have within the virtual machine. In order to achieve this, let's add the following line to our Vagrant file. Some versions of Vagrant will automatically include this line but commented it out, so we may just need to remove the comment (# character from the start of the line):

```
config.vm.network :forwarded_port, guest: 80, host: 8080
```

Before we run our Puppet provisioner to install our LAMP stack, we should instruct Vagrant to run the `apt-get update` command on the virtual machine. Without this, it isn't always possible to install new packages:

```
config.vm.provision :shell, :inline => "apt-get update"
```

As we will be putting our Puppet modules and manifests in a provision folder, we need to configure Vagrant to use the correct folders for our Puppet manifests and modules as well as the default manifest file. Adding the following to our Vagrant file will do this for us:

```
config.vm.provision :puppet do |puppet|
    puppet.manifests_path = "provision/manifests"
    puppet.module_path = "provision/modules"
    puppet.manifest_file  = "default.pp"
end
```

Creating the Puppet manifests

Let's start by creating some folders for our Puppet modules and manifests by executing the following commands:

```
mkdir provision
cd provision
mkdir modules
mkdir manifests
```

For each of the modules we want to create, we need to create a folder within the `provision/modules` folder for the module. Within this folder, we need to create a `manifests` folder, and within this our Puppet manifest file, `init.pp`. Structurally, this looks something, as follows:

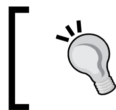
```
|-- provision
|   |-- manifests
|   |     -- init.pp
|    -- modules
 -- Vagrantfile
```

Installing Apache

Let's look at what is involved in installing Apache (this would be in the file, `provision/modules/apache/init.pp`). First, we need to ensure the `Apache2` package is installed:

```
class apache {

  package {"apache2":
    ensure => present
  }
```



Note that we have not closed the curly bracket for the `apache` class. That is because this is just the first snippet of the file; we will close it at the end.

Next, we should, within the default Apache web root (`/var/www`), create a symlink which points a `src` folder to our projects `src` folder (this is within the shared folder that Vagrant automatically creates). This needs to be done once Apache is installed (to ensure the `/var/www` folder is present), as follows:

```
  file { '/var/www/src':
      ensure => 'link',
      target => '/vagrant/src',
      require => Package['apache2']
  }
```

Because we want to change our default Apache configuration file, we should update the contents of the Apache configuration file with one of our own (this will need to be placed in the `provision/modules/apache/files` folder in a file called `default`).

```
  file { '/etc/apache2/sites-available/default':
    source => 'puppet:///modules/apache/default',
    owner => 'root',
```

```
    group => 'root',
    require => Package['apache2']
  }
```

Because we want to reload Apache once some of the PHP packages have been installed, we can tell the Apache service to subscribe to these packages. Once they are installed, Apache will restart. The following code will subscribe the Apache service to these other packages and trigger the restart when they are installed:

```
  service { "apache2":
    require => Package["apache2"],
    subscribe => [File['/etc/apache2/sites-available/default'],
      Package['php5', 'php5-mysql', 'php5-dev', 'php5-curl',
        'php5-gd', 'php5-imagick', 'php5-mcrypt', 'php5-memcache',
          'php5-mhash', 'php5-pspell', 'php5-snmp', 'php5-xmlrpc',
            'php5-xsl', 'php-pear', 'libapache2-mod-php5']]
  }
```

We might also want to support file uploads within our project, so let's create an `uploads` folder which is owned by the Apache user (`www-data`) and can be written to (`chmod: 0777`):

```
  file{ "/var/www/uploads":
    ensure => "directory",
    owner  => "www-data",
    group  => "www-data",
    mode   => 777,
    require => Package['apache2']
  }

}
```

Two changes we need to make to the Apache configuration are as follows:

- Create an `alias` folder which points to our `uploads` folder (we need to create an `alias`, because this folder is outside of the web project files)
- Set the document root to the symlink we created earlier

This is the file we need to call `default` and save within the `provision/modules/apache2/files` folder:

```
<VirtualHost *:80>
        ServerAdmin webmaster@localhost

        Alias /uploads /var/www/uploads

        DocumentRoot /var/www/src
        <Directory />
                Options FollowSymLinks
                AllowOverride None
        </Directory>

        ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
        <Directory "/usr/lib/cgi-bin">
                AllowOverride None
                Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
                Order allow,deny
                Allow from all
        </Directory>

        ErrorLog /var/log/apache2/error.log

        # Possible values include: debug, info, notice, warn, error,
crit,
        # alert, emerg.
        LogLevel warn

        CustomLog /var/log/apache2/access.log combined

    Alias /doc/ "/usr/share/doc/"
    <Directory "/usr/share/doc/">
        Options Indexes MultiViews FollowSymLinks
        AllowOverride None
        Order deny,allow
        Deny from all
        Allow from 127.0.0.0/255.0.0.0 ::1/128
    </Directory>

</VirtualHost>
```

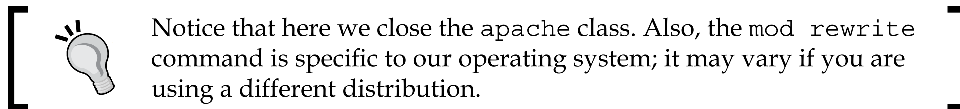
Enable an Apache rewrite module

The Puppet module we need to create to enable the rewrite module is fairly basic; we simply need to run a command, which installs the module. The code requires that we already have Apache installed:

```
class modrewrite{

  exec { 'enabledmodrewrite':
    command => '/usr/sbin/a2enmod rewrite',
    require => Package['apache2']
  }

}
```



Notice that here we close the `apache` class. Also, the `mod rewrite` command is specific to our operating system; it may vary if you are using a different distribution.

Installing MySQL

Installing MySQL is also fairly straightforward, we just want to install a few related packages. The following code should be placed in the file `provision/modules/mysql/init.pp`:

```
class mysql {

  package { "mysql-server":
    ensure => present
  }

  package { "mysql-client":
    ensure => present
  }

  package { "libmysqlclient15-dev":
    ensure => present
  }
}
```

Installing PHP

To install PHP we need to install a range of related packages including the Apache PHP module. This would be in the file `provision/modules/php/init.pp`:

```
class php {

  package { "php5":
    ensure => present
  }

  package { "php5-mysql":
    ensure => present
  }

  package { "php5-dev":
    ensure => present
  }

  package { "php5-curl":
    ensure => present
  }

  package { "php5-gd":
    ensure => present
  }

  package { "php5-imagick":
    ensure => present
  }

  package { "php5-mcrypt":
    ensure => present
  }
```

```
    package { "php5-memcache":
      ensure => present
    }

    package { "php5-mhash":
      ensure => present
    }
    package { "php5-pspell":
      ensure => present
    }

    package { "php5-snmp":
      ensure => present
    }

    package { "php5-xmlrpc":
      ensure => present
    }

    package { "php5-xsl":
      ensure => present
    }

    package { "php5-cli":
      ensure => present
    }

    package { "php-pear":
      ensure => present
    }

    package { "libapache2-mod-php5":
      ensure => present,
      require => [Package[php5], Package[apache2]]
    }
  }
```

Installing e-mail

Because some of our projects might involve sending e-mails, we should install e-mail sending services on our virtual machine, and we should set these to automatically run when the server boots up, as follows:

```
class mail {

  package { "postfix":
    ensure => present
  }

  package { "mailutils":
    ensure => present
  }

  exec { 'autostartmail':
    command => '/usr/sbin/update-rc.d postfix defaults',
    require => Package['postfix']
  }

}
```

Default manifest

Finally, we need to pull these modules together and install them when our machine is provisioned. To do this, we simply add the following to our `default.pp` manifest file in the `provision/manifests` folder, as follows:

```
import "apache"
include apache
import "modrewrite"
include modrewrite
import "php"
include php
import "mysql"
include mysql
import "mail"
include mail
```

Launch the VM

In order to launch our new virtual machine, we simply need to run the following command:

```
Vagrant up
```

As per *Chapter 5*, *Provisioning with Vagrant using Puppet and Chef*, we should now see our VM boot, and that the various Puppet phases execute. If all goes well, we should see no errors in this process.

Summary

In this chapter, we learned about the steps involved in creating a brand new Vagrant project, configuring it to integrate with our host machine, and set up the standard LAMP stack using the Puppet provisioning tool.

Index

A

add subcommand 21
admin group
about 84
creating 85
Vagrant user, adding to 85
Ansible
about 64
URL 65
using 65
Apache
about 6, 89
configuration 92
installing 91, 92
Apache class 34 37
Apache installation, in Chef
about 49
Apache package, installing 50
Apache service, executing 51
package manager, updating 50
Apache installation, in Puppet
about 37
Apache package, installing 38
Apache service, executing 39
package manager, updating 38
Apache rewrite module
enabling 94
Apache service
executing 39
apply command 60
apply subcommand 45
apt-get update command 38, 50
authorized_hosts file 86
auto-running commands 28, 29

B

base box
creating 78
exporting 87
importing 16-18
used, for creating Vagrant project 16-18

C

cache
clean up 87
Chef
about 6-8, 47, 48
Apache, installing 49
Chef solo 63
client/server mode 64
cookbooks, creating with 48
cron, managing 54
cron resource 49
execute resource 49
file management 51
file resource 49
group resource 49
groups, creating 55
groups management 55
installing 86
provisioning 62
recepies, creating with 48
resources 49
sudoers file, updating 56
template resource 49
user resource 49
users, creating 55, 56
users management 55
using, with Vagrant 57

Chef Client 64
Chef cookbooks
  using 56
Chef recipes 48
Chef Roles
  about 63
  reference link 63
Chef Server 64
Chef solo
  about 48, 63
  using 63
chef-solo command 57
classes, Puppet 33, 34
cleanup 87
client/server mode, Chef 64
client/server mode, Puppet 62
command line
  execution 66
commands, Chef
  running 54, 55
commands, Puppet
  running 42, 43
composer.lock file 55
configuration, Apache 92
cookbook
  about 48
  creating, with Chef 48
creates parameter 42, 54
cron, Chef
  managing 54
cron management, Puppet 42
cron resource, Chef 49
cron resource, Puppet 35
cron resource type, Puppet 42

D

default manifest 97
default Puppet manifest 34, 35
destroy subcommand 26

E

e-mail
  installing 97
exec command 36, 37, 42-44, 54-56
exec resource, Puppet 35
Exec resource type, Puppet 42
execute resource, Chef 49

F

file management, Chef
  about 51
  file, copying 51, 52
  folders, creating 53
  multiple folders, creating 53
  symlink, creating 52
file management, Puppet
  about 39
  file, copying 39
  folders, creating 41
  multiple folders, creating 41
  symlink, creating 40
file resource, Chef 49
file resource, Puppet 35
functionalities, Vagran 23, 24

G

Git 6
group resource, Chef 49
group resource , Puppet 35
groups, Chef
  creating 55
groups, Puppet
  creating 43
Guest Additions, VirtualBox
  about 83, 84
  installing 84
guest machine
  controlling 23
  integrating, with host machine 27

H

host machine
  integrating, with guest machine 27

I

init subcommand 16
insecure private key pair 86
insecure public key pair 85, 86
installation, Apache 91, 92

installation, e-mail 97
installation, MySQL 94
installation, PHP 95
installation, Puppet 86
installation, Ruby gems 87
installation, Vagrant 13, 14
installation, VirtualBox 8-12
installation, Chef 87
inventory file 65

L

LAMP 89
LAMP server project 38
Linux 8, 89
list subcommand 22

M

Mac OS X 8
MAMP 7
manifests folder 60
manifests, Puppet 33
modules, Puppet
about 33
classes 33
multi-machine project
destroying 75
multiple provisioners
using 65
multiple virtual machines
connecting to, SSH used 71, 72
defining 70
networking 72, 73
provisioning 74
using, with Vagrant 70
MySQL
about 89
installing 94

N

Network File System 28
networking 28
Nginx 6

O

Opscode
URL, for resource types 49
Oracle VirtualBox 8

P

package resource, Chef 49
package resource, Puppet 35
package subcommand
about 87
reference link 87
parameters 36
PHP
about 89
installing 95
playbook 65
port forwarding 27
prerequisites, Vagrant installation 8
Providers 6
provision command 60
provisioners 86
provisioning
about 31, 32
overriding 66
overriding, via command line 66
within Vagrant 59
provisioning options, Vagrant 28
Puppet
about 6-8, 31-33
Apache, installing 37
client/server mode 62
commands, runnning 42, 43
cron management 42
file management 39
groups, creating 43
groups management 43
installing 86
manifests 33
modules 33
provisioning 60-62
refreshonly parameter 45
resource execution ordering 37
resource requisites 36
resources 35

standalone mode 60, 61
subscribe parameter 44
sudoers file, updating 44
URL 36
users, creating 43
users management 43
using, with Vagrant 45
Puppet Forge
about 32, 45
URL 45
Puppet labs
URL, for info 86
Puppet manifests
about 32
creating 90, 91
Puppet modules 45
PuTTY 26

R

recepies
creating, with Chef 48
refreshonly parameter, Puppet 44
remove subcommand 22
repackage subcommand 23
resource execution ordering, Puppet 37
resource requisites, Puppet 36
resources, Chef
about 49
common functionality 56
resources, Puppet 35
resource types, Chef 49
resource types, Puppet
cron 35
exec 35
file 35
group 35
package 35
service 35
user 35
rewrite module, Apache
about 94
enabling 94
Ruby 8
Ruby Domain Specific Language 6, 32
Ruby files 48
Ruby gems
installing 87

S

service resource, Chef 49
service resource, Puppet 35
specific values
setting up 83
SSH
about 64
provisioning 65
SSH ports
multiple virtual machines, connecting over 71, 72
SSH provisioning 28
standalone mode, Puppet 60, 61
subscribe parameter, Puppet 44
Subversion 6
sudoers file
about 56, 85
updating 44, 56
URL, for info 85
supervisord module 34
symlink
creating 40, 52
Synced folders 27, 28

T

template resource, Chef 49

U

Ubuntu Lucid 23
Ubuntu Precise 23
Ubuntu Server Version 13.04
URL 78
up subcommand 26
user resource, Chef 49, 55
user resource, Puppet 35, 43
users, Chef
creating 55, 56
users, Puppet
creating 43
users resource type, Puppet 43

V

Vagrant
about 6
Ansible, using 65
benefits 7
Chef, provisioning 62
Chef, used with 57
connecting, to multiple virtual machines over SSH 71, 72
connecting, to virtual machine over SSH 26
features 31
functionalities 24
installing 13, 14
multiple virtual machines, using with 70
provisioners 86
provisioning options 28
Puppet, provisioning with 60
Puppet, using with 45
SSH provisioning 28
URL 6
URL, for documentation 77
URL, for downloading installer 13
Vagrant authentication
about 84
admin group 84
insecure private key pair 86
insecure public key pair 85
sudoers file 85
Vagrant user 84
Vagrant boxes
adding 21
listing 22
managing 20
removing 22
repackaging 23
searching 23
Vagrant file 6, 16, 23, 64
vagrant halt command 25
vagrant init command 19
used, for creating Vagrant project 19
Vagrant installation
prerequisites 8
Vagrant project
creating 15, 89, 90
creating, base box used 16-18
creating, vagrant init command used 19
creating, without importing base box 19
Vagrant project, creating
base box, importing 16-18
vagrant resume command 25
vagrant ssh command 26, 72
vagrant ssh database 72
vagrant suspend command 25
vagrant up command 23
Vagrant user
about 84
adding, to admin group 85
creating 84
Version Control System 6, 26
VirtualBox
about 6-8
creating 78-83
installing 8-12
URL, for downloading installer 8
virtualization 31
virtualized environment 5
virtual machine (VM)
halting 25
launching 98
powering up 23
resuming 25
suspending 25
Vagrant, connecting 26
VMDK (Virtual Machine Disk) 80
VMware Fusion 8

W

WAMP 7
Windows 8
Workstation 8

Thank you for buying

Creating Development Environments with Vagrant

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 licences, 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.

[PACKT] open source
community experience distilled
PUBLISHING

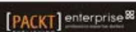
OpenStack Cloud Computing Cookbook, Second Edition: RAW

ISBN: 978-1-78216-758-7 Paperback: 310 pages

Over 100 recipes to successfully set up and manage your OpenStack cloud environments with complete coverage of Nova, Swift, Keystone, Glance, Horizon, Quantum, and Cinder

1. Updated for OpenStack Grizzly
2. Learn how to install, configure, and manage all of the OpenStack core projects including new topics like block storage and software defined networking
3. Learn how to build your Private Cloud utilizing DevOps and Continuous Integration tools and techniques

---



Citrix XenApp Performance Essentials

ISBN: 978-1-78217-044-0 Paperback: 98 pages

A practical guide for tuning and optimizing the performance of XenApp farms using real world examples

1. Design a scalable XenApp infrastructure
2. Monitor and optimize server performance
3. Improve end user experience
4. Tune the farm for WAN connections
5. Real world examples, ready-to-use suggestions, and best practices

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
PUBLISHING
community experience distilled

vSphere High Performance Cookbook

ISBN: 978-1-78217-000-6 Paperback: 240 pages

Over 60 recipes to help you improve vSphere performance and solve problems before they arise

1. Troubleshoot real-world vSphere performance issues and identify their root causes
2. Design and configure CPU, memory, networking, and storage for better and more reliable performance
3. Comprehensive coverage of performance issues and solutions including vCenter Server design and virtual machine and application tuning

---

Microsoft System Center Virtual Machine Manager 2012 Cookbook

ISBN: 978-1-84968-632-7 Paperback: 342 pages

Over 60 recipes for the administration and management of Microsoft System Center Virtual Machine Manager 2012 SP1

1. Create, deploy, and manage Datacentres, Private and Hybrid Clouds with hybrid hypervisors by using VMM 2012 SP1, App Controller, and Operations Manager.
2. Integrate and manage fabric (compute, storages, gateways, networking) services and resources. Deploy Clusters from bare metal servers.
3. Learn how to use VMM 2012 SP1 features such as Windows 2012 and SQL 2012 support, Network Virtualization, Live Migration, Linux VMs, Resource Throttling, and Availability.

Please check **www.PacktPub.com** for information on our titles