O'REILLY®

Learning Puppet 4

A GUIDE TO CONFIGURATION MANAGEMENT AND AUTOMATION

Jo Rhett

learning puppet
Book Subtitle

Jo Rhett

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Learning Puppet 4
by Jo Rhett

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Brian Anderson
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2015-06-12 First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491907634* for release details.

978-1-491-90763-4

[FILL IN]

Table of Contents

Preface ix

Introduction xiii

Foreword xvii

Part I. Controlling with Puppet Apply

1. Thinking Declarative 21
Handling Change 21
Idempotence 22
Declaring Final State 23
Conclusion 24

2. Creating a Learning Environment 25
Installing Vagrant on Mac 26
Installing Vagrant on Windows 28
Starting a Command Prompt 36
Downloading a Box 37
Initialize Vagrant System 37
Initialize Non-Vagrant System 39
Choosing a Text Editor 40
On the Virtual System 40
On your Desktop 41
Conclusion 42

3. Installing Puppet. 43
Adding the Package Repository 43
What is a Package Collection? 44
Installing the Puppet Agent 44
Reviewing Dependencies 45
Reviewing Puppet4 Changes 46
Linux and Unix 46
Windows 48
Making Tests Convenient 48
Conclusion 49

4. Writing Manifests. 51
Implementing Resources 51
Applying a Manifest 52
Declaring Resources 53
Viewing Resources 54
Executing Programs 55
Managing Files 56
Declarative Review 58
Testing Yourself 59
Conclusion 59

5. Using Puppet Configuration Language. 61
Defining Variables 61
Defining Numbers 62
Using Variables in Strings 63
Limiting Problems with Brackets 63
No Redefinition 64
Finding Facts 65
Retrieving Values 66
Avoiding Reserved Words 67
Modifying with Operators 68
Order of Operations 69
Using Conditional Operators 70
Creating Regular Expressions 71
Evaluating Conditional Expressions 72
Building Lambda Blocks 74
Looping through Iterations 75
Each 77
Filter 78
Map 79
Reduce 80

Slice 81
With 82
Captures-Rest Parameters 82
Summary 82
Conclusion 82

6. Controlling Resource Processing. 85
Adding Aliases 85
Preventing Action 86
Auditing Changes 86
Defining Loglevel 86
Limiting by Tags 87
Limiting by Schedule 88
Defining Resource Defaults 89
Conclusion 89

7. Expressing Relationships. 91
Managing Dependencies 92
Referring to Resources 92
Ordering Resources 93
Triggering Refresh Events 93
Chaining Resources with Arrows 95
Processing with Collectors 95
Understanding Puppet Ordering 96
Conclusion 97

8. Upgrading Puppet 3 Manifests. 99
Validating Numbers 99
File Modes are not Numbers 100
Using Hash and Array Literals 100
Adding Else to Unless 101
Chaining Assignments 101
Expressions Can Stand Alone 101
Chaining Expressions with a Semicolon 102
Calling Functions in Strings 102
Improved Error Reporting 102
Avoiding Upgrade Problems 103
Deprecations 104

9. Conclusion of Part I. 105
Best Practices for Writing Manifests 105
Continued Learning 106

Part II. Creating Puppet Modules

10. Creating a Test Environment . . . 109
Verifying the Production Environment 109
Creating a Test Environment copy 110
Changing the Base Module Path 110
Skipping Ahead 111

11. Separating Data from Code . . . 113
Introducing Hiera 113
Creating Hiera Backends 114
Hiera Data in YAML 114
Hiera Data in JSON 115
Hiera Data in Puppet 116
Puppet Variable and Function Lookup 116
Configuring Hiera 116
Backends 117
Backend Configuration 117
Logger 118
Hierarchy 118
Merge Behavior 119
Complete Example 120
Doing Hiera Lookups in a Manifest 120
Testing Hiera Lookups 121

12. Using Modules . . . 123
Finding Modules 123
Puppet Forge 123
Public GitHub Repositories 124
Internal Repositories 125
Evaluating Module Quality 125
Puppet Supported 126
Puppet Approved 127
Quality Score 128
Community Rating 129
Installing Modules 130
Installing from a Puppet Forge 130
Installing from GitHub 131
Testing a Single Module 131
Defining Config with Hiera 132
Executing Multiple Modules with Hiera 133

Examining a Module 135
Reviewing Modules 135

13. Designing a Custom Module. 137
Choosing a Module Name 137
Avoiding Reserved Names 138
Generating a Module Skeleton 138
Modifying the Default Skeleton 139
Understanding Module Structure 139
Creating a Class Manifest 140
What is a Class? 140
Accepting Input 141
Validating Input with Types 142
Valid Types 143
Accepting Values 144
Testing Values 145
Matching Regular Expressions 147
Declaring Resources 147
Using Hiera Data 148
Sharing Files 149
Parsing Templates 151
Common Syntax 152
Using Puppet EPP Templates 152
Using Ruby ERB Templates 155
Creating Readable Templates 157
Building Subclasses 157
Understanding Variable Scope 159
Reusing Defined Types 160
Calling Other Modules 161
Sourcing a Common Dependency 161
Using a Different Module 163
Ordering Dependencies 164
Containing Classes 165
Documenting the Module 166
Learning Markdown 166
Updating README.md 167
Creating CHANGELOG.md 169
Documenting the Classes and Types 169
Peeking Beneath the Hood 174
Best Practices for Module Design 175
Modules Review 176

14. Testing Modules. 177
Installing Dependencies 177
Installing Ruby 177
Installing Gem Bundler 178
Installing Spec Helper 178
Preparing Your Module 179
Defining Fixtures 179
Defining Tests 180
Defining Main Class 181
Passing Valid Parameters 182
Failing Invalid Parameters 183
Adding an Agent Class 184
Using Hiera Input 184
Defining Parent Class Parameters 185
Improve Testing with Custom Skeletons 186
Simplifying with Tools 187
Puppet-Retrospect 187
Finding Documentation 188
Testing Modules Review 188

15. Extending Modules with Plugins. 189
Adding Custom Facts 189
External Facts 190
Custom (Ruby) Facts 192
Understanding Implementation Issues 197
Defining Functions 197
Puppet Functions 197
Ruby Functions 198
Using Custom Functions 202
Providing Data in Modules 202
Module Plugins Review 202
Requirements for Module Plugins 203

16. Publishing Modules. 205
Packaging a Module 205
Uploading a Module to the Puppet Forge 206
Publishing a Module on GitHub 207
Automating Module Publishing 209
Getting Approved Status from Puppet Labs 209

Preface

This book is a work in progress – new chapters will be added as they are written. We welcome feedback – if you spot any errors or would like to suggest improvements, you can submit errata or send email directly to jrhett@netconsonance.com.

Similarly, we've made a survey available for you to share your feedback, here.

This book will teach you how to install and use Puppet, a configuration management system. It will introduce you to how Puppet works, and how Puppet provides value to you. You'll learn how to setup a testing environment you can use to learn Puppet, then keep and evolve as your Puppet knowledge grows. You'll learn how to declare and implement configuration policy for hundreds of nodes.

This book covers modern best practices for Puppet 3 and Puppet 4. You'll find tips throughout the book labeled **Best Practice**.

You'll learn how to update an existing Puppet 2 or Puppet 3 installation for the increased features and improved parser of Puppet 4. You'll learn how to run Puppet services over IPv6 protocol.

Most important of all, this book will cover how to scale your Puppet installation to handle thousands of nodes. You'll learn multiple strategies for handling diverse and heterogenous environments, and reasons why each of these approaches may be appropriate or not for your needs.

Who this book is for

This book is primarily aimed at System Administrators and Operations or DevOps Engineers. If you are responsible for development or production nodes, this book will provide you with immediately useful tools to make your job easier than ever before. If you run a high-uptime production environment, you're going to learn how Puppet

can enforce your existing standards throughout the implementation. Within a week you'll wonder how you ever got along without it.

No matter what you call yourself, if you feel that you spend too much time managing computers then this book is for you. You'd like to get it done faster so you can focus on something else. You'd like to do it more consistently, so that you don't have to chase down one-off problems in your reports. Or you've got some new demands that you're looking for a way to solve. If any of these statements fit, you will find Puppet to be one of the best tools in your toolbox.

What to expect from me

This book will not be a heavy tome filled with reference material irrelevant to the day to day system administrator---exactly the opposite. Throughout this book we will never stray from one simple goal: We focus all our efforts on how Puppet can help you do something faster or better than ever before.

This book will never tell you to run a script and not tell you what it does, or why. I hate modeling systems to determine what an installation script did, and I won't do this to you. In this book you will build up the entire installation by hand. Every step you take will be useful to you in a production deployment. You'll know where every configuration file lives. You'll learn every configuration parameter and what it means.

By the time you have finished this book, you'll know how Puppet works inside and out. You will have the tools and knowledge to deploy Puppet seamlessly throughout your environment.

What you will need

You may use any modern Linux, Mac, or Windows system and successfully follow the hands-on tutorials in this book.

While there are some web dashboards for Puppet, the process of configuring and enabling Puppet, and utilization of the agent will be performed through the command line. We will help you install any necessary software.

A beginner to system administration can follow every tutorial in this book. Any experience with scripts, coding, or configuration management will enhance what you can get out of this book, but is not necessary. It is entirely possible to deploy Puppet to manage complex environments without writing a single line of code.

Part II: Puppet Modules documents how to build custom modules for Puppet. Most Puppet modules can be created using the Puppet configuration language that will be taught in this book. When you've become an expert in building Puppet modules, you may want to add new functions to the Puppet configuration language. New functions

are created in pure Ruby. Reference materials like http://shop.oreilly.com/product/9780596529864.do[Learning Ruby] can be helpful when creating new functions for use in a custom Puppet module.

What you'll find in this book

The Introduction provides an overview of what Puppet does, how it works, and why you may want to use it.

Part I: Getting Started will get you running with a working Puppet installation. You will learn how to write declarative Puppet policies to produce consistency in your systems. This will also cover the changes in the language that Puppet 4 has brought.

Part II: Puppet Modules will introduce you to Puppet modules, the building blocks used for Puppet policies. You will learn where to find Puppet modules. You'll learn how to distinguish Puppet Labs-provided and Puppet Approved modules. More importantly, you'll learn how to build, test, and publish your own modules.

Part III: Puppet Master will help you install the Puppet master. You will learn the integrated components that make up the Puppet infrastructure. You'll install and configure each in a manner suitable for your specific environment.

Part IV: Advanced Puppet will review real-life deployment considerations. You will learn about ways to scale puppet to thousands or tens of thousands of nodes. You'll learn how to manage the infrastructure that Puppet depends upon using Puppet.

Part V: Puppet Ecosystem will show you infrastructure that ties together with, enables, or supports your Puppet installation. While each of these is worthy of their own book, you'll install and configure each of these to provide immediate value in your Puppet environment.

This won't be a test environment for training that doesn't match your real concerns; instead you'll perform real operations on hosts that match your production environment. You'll see how easy it is to deploy Puppet, and exactly how powerful the tools it provides are. You'll receive hands-on tips and experiences from years of experience deploying, scaling, and tuning Puppet environments.

Puppet has an active developer and user community. Using Community Modules directs you to online repositories of modules built by others, as well as concrete examples of how to use other's modules in your environment.

How to Use this Book

This book provides explicit instructions for configuring and using Puppet from the command line without the use of external tools beyond your favorite text editor.

The book will help you create Puppet manifests and Puppet modules which utilize every feature of Puppet. You will find it easy to create configuration policies to handle your specific needs from the examples in this book. Yes, with just your text editor.

The book documents a Puppet module which can be used to maintain Puppet servers and agents across dozens of environments. This module, provided in Part II: Puppet Modules, could be easily adjusted to manage both your Puppet servers, and the Puppet agents on each node across all of your environments.

In Part V: Puppet Ecosystem, we will introduce related and integrated systems that supplement and utilize Puppet. Some of these systems provide a web server interface for viewing or managing the status and history of nodes managed by Puppet.

IPv6 Ready

Every example with IP addresses will include both IPv4 and IPv6 statements. If you're only using one of these protocols you can ignore the other. Puppet will happily use any combination of them. More details about complex IPv6 setups will be covered in IPv6 Dual-Stack Environments.

Acknowledgements

I owe significant gratitude to Luke Kaines, who conceived of Puppet, and continues to direct its growth in Puppet Labs. I was working on CFengine with him and others when he left that effort to create Puppet. His vision and foresight made all of this possible.

I owe a drink and many thanks to the many people who provided input and feedback on the book during the writing process, including but definitely not limited to the technical reviewers:

- Chris Barbour, Taos Mountain

And finally, I'd like to thank my O'Reilly editor, Brian Anderson, who gave me excellent guidance on the book and was a pleasure to work with.

Jo Rhett, DevOps Architect, Net Consonance

Introduction

What is Puppet?

Puppet brings computer systems into compliance with a policy you design. Puppet manages configuration data on these systems, including users, packages, processes, services; any component of the system you can define. Puppet can manage complex components to ensure compliance with the policies you write.

Puppet can ensure configuration consistency across thousands of servers. Puppet utilizes node-specific data to tune the policy appropriately for each system.

As an administrator, you will utilize the Puppet configuration language to declare the final state of your systems. Thus, we describe Puppet as "declarative."

Why Declarative

When analyzing hand-built automation systems, you'll invariably find commands like the following:

```
$ sed -i -e 's/regex/replacement/eg' filename
```

This command takes a file and replaces data within the file with a processed result. This works properly the first time you run it. However if the same operation is run again, the result changes. This isn't a desirable effect in configuration management.

Language which describes the actions to perform is called "procedural". It defines procedures that should be followed to change the target.

When managing computer systems, you want the operations applied to be **idempotent**, where the operation achieves the same results every time it executes. This allows you to apply and re-apply the configuration policy and always achieve the desired state.

In order for a configuration state to be achieved no matter the conditions, it is essential that the configuration language avoid describing the actions involved to achieve the desired state. Instead, the configuration language should describe the desired state, and leave the actions up to the interpreter. Language which declares the final state is called "declarative."

Declarative language is much easier to read, and less prone to breakage due to environment differences. Puppet was designed to achieve consistent and repeatable results. Every time Puppet evaluates the state of the node, it will bring the node to a state consistent with the configuration policy.

How Puppet Works

On any node you control is an application named **Puppet agent**. The agent evaluates and implements *Puppet manifests*, or files containing Puppet configuration language that declare the desired state of the node. The agent evaluates the state of each component described in a manifest, and determines whether or not any change is necessary. If the component needs to be changed, the agent makes the requested changes and logs the event.

If Puppet is configured to utilize a centralized Puppet master, Puppet will send the node's data to the master, and receive back a pre-compiled *catalog* containing only the node's specific policy to enforce.

Now you might be thinking to yourself, "What if I only want the command executed on a subset of nodes?" Puppet provides many different ways to classify and categorize nodes to limit which resources should be applied to which nodes. You can use node facts like hostname, operating system, node type, puppet version, and many others. Best of all, new criteria custom to your environment can be easily created.

The Puppet agent evaluates the state of only one node. In this model you can have agents on tens, hundreds, or thousands of nodes evaluating their catalogs and implementing changes on their nodes at exactly the same time. The localized state machine ensures a scalable and fast parallel execution environment.

Why Use Puppet

As we have discussed above, Puppet provides a well-designed infrastructure for managing state of many nodes simultaneously. Here are a few reasons to use it:

- Facter provides Puppet with local data to customize the policy for each specific node: hundreds of values specific to the node including hostname, operating system, memory, networking configuration, and many node-specific details.
- Puppet agents can handle OS-specific differences, allowing you to write a single manifest which will work on different operating systems.

- Puppet agent can be invoked with specific tags, allowing a filtered run thats only perform operations which match those tags during a given invocation.
- Puppet agents report back success, failure, and specific return codes for each run.
- Puppet uses a decentralized approach were each node evaluates and executes their own Puppet catalog separately. No node is waiting for another node to complete.
- Orchestration systems such as the Marionette Collective (MCollective) can invoke and control the Puppet agent for instantaneous large-scale changes.

In Part II: Puppet Modules you will create a module that uses Puppet to install and configure the Puppet agent. This kind of recursion is not only possible but common.

In Part III: Puppet Master you will learn how to use Puppet masters and Puppet Server to offload and centralize manifest compilation, report processing, and backup of changed files.

In Part V: Puppet Ecosystem you will use MCollective to orchestrate immediate changes with widespread Puppet agents.

Puppet provides a flexible framework for policy enforcement that can be customized for any environment. After reading this book and using Puppet for a while, you'll be able to tune your environment to exactly your needs. Puppet's declarative language not only allows but encourages creativity.

Time to Get Started

As we proceed, this book will show you how Puppet can help you do more, do it faster, and more consistently than ever before. You'll learn how to extend Puppet to meet your specific needs:

1. You'll install Puppet and get it working seamlessly to control files, packages, services, and the Puppet daemon.
2. You'll discover an active community of Puppet developers who develop modules and other Puppet plugins on the Puppet Forge and GitHub.
3. You'll build your own custom Fact. You'll use this fact within your Puppet manifest to handle something unique to your environment.
4. You'll build your own custom Puppet module. You'll learn how to test the module safely prior to deploying in your production environment.
5. You'll learning how to package your module and upload it to a Puppet Forge.
6. You'll learn how to configure a Puppet Server, allowing you to provide Puppet services across the campus or around the globe.
7. You'll tour through the ecosystem of components which utilize, extend, and enhance Puppet within your environment.

By the time you finish this book you will understand not just how powerful Puppet is, but you'll know exactly how Puppet works. You'll have the knowledge and understanding to debug problems within any part of the infrastructure. You'll know what to tune as your deployment grows. You'll have a resource to use for further testing as your knowledge and experience expands.

It's time to get declarative.

Foreword

There will be something awesome here in a later revision.

PART I

Controlling with Puppet Apply

In this part you'll learn about the Puppet configuration language and how to think in a declarative manner. You'll set up a testing environment you can use to learn Puppet while reading this book. You'll be able to continue to use this setup to develop and test your Puppet code long after you have finished this book.

You will install Puppet, and create your first Puppet manifests. You'll learn how to utilize resources, how to associate them, and how to order and limit the changes upon them.

When you finish this part of the book, you'll have a solid understanding of the Puppet configuration language. You'll have written and tested your own Puppet manifests. You'll have a solid foundation of modern Best Practices for Puppet coding and style.

CHAPTER 1

Thinking Declarative

If you have any experience with conventional programming languages, or writing shell scripts, you are used to making change by defining the operations which should be performed. Code which defines each operation, each procedure to be executed, and the order in which to execute them, is known as Procedural Programming code.

While it can be useful to have a background in procedural programming, a common mistake is to attempt to use Puppet to make changes in a procedural fashion. The very best thing you can do is forget everything about procedural programming.

If you are new to programming, don't feel intimidated. People without a background in procedural programming can often learn good Puppet practices faster.

Writing good Puppet manifest is done with declarative programming. When it comes to maintaining configuration on systems, you'll find declarative programming to be easier to create, easier to read, and easier to maintain. Let's show you why.

Handling Change

The reason that you need to cast aside procedural programming is to handle change better.

When you write code that performs a sequence of operations, that sequence will make the desired change the first time it is run. If you run the same code the second time in a row, the same operations will either fail, or will create a different state than desired. Here's an example:

```
# useradd -u 1001 -g 1001 -c "Joe User" -m joe
# useradd -u 1001 -g 1000 -c "Joe User" -m joe
useradd: user 'joe' already exists
```

So then you need to change the code to handle that situation.

```
#!/bin/bash
USERNAME=$0
getent passwd joe > /dev/null 2> /dev/null
if [ $? -ne 0 ]
then
    useradd -u 1001 -g 1000 -c "Joe User" -m joe
fi
```

Okay, that's at six lines of code and all we've done is ensure that the username isn't already in use. What if we need to check to ensure the UID is unique, the GID is valid, that password expiration is set? Well, I think you know this is going to be a very long script even before we adjust it to ensure it works properly on multiple operating systems.

This is why we say that procedural programming doesn't handle change very well. It takes a lot of code to cover every situation you need to test.

Idempotence

When managing computer systems, you want the operations applied to be **idempotent**, where the operation achieves the same results every time it executes. Idempotence allows you to apply and re-apply a configuration manifest and always achieve the desired state.

In order for procedural code to be idempotent, it needs to have instructions for how to compare, evaluate, and implement not just every resource, but each attribute of the resource. As you saw in the previous section, this will quickly become ponderous and difficult to maintain.

Idempotent: operations in mathematics and computer science that can be applied multiple times without changing the result beyond the initial application. It literally means *(the quality of having) the same power*, from latin roots **idem** + **potent** (same + power)[1]. Here are some examples of idempotent and not math and code:

`any number^1`	**idempotent**	Any number to the power of 1 is the same (implicit definition)
`variable = variable * 2`	**not idempotent**	Will double every time
`variable = variable * 2 / 2`	**idempotent**	remains the same value
`echo "Today is a good day!" >> /some/file`	**not idempotent**	file will keep growing

1 http://www.jeremy-gunawardena.com/papers/intro.pdf

```
echo "Today is a good
day!" > /some/file
```

idempotent 2nd run file will same content every time

The simplistic final example avoids having to compare the state of the item by simply overwriting it every time. This means it will only be idempotent on the 2nd and following invocations. This only works in a limited set of situations. Most changes require evaluation to determine what changes are necessary.

Declaring Final State

In order for a configuration state to be achieved no matter the conditions, it is essential that the configuration language avoid describing the actions involved to achieve the desired state. Instead, the configuration language should describe the desired state, and leave the actions up to the interpreter. Language which declares the final state is called *declarative*.

Rather than writing extensive procedural code to handle every situation, it is much simpler to declare what you want the final state to be. Rather than dozens of lines of comparison, the code reflects only the final state that the resource (in this example a user account) should be. Here we will introduce you to your first bit of Puppet configuration language, a resource declaration for the same user we created above.

```
user { 'joe':
  ensure     => present,
  uid        => '1001',
  gid        => '1000',
  comment    => 'Joe User',
  managehome => true,
}
```

As you can see above, the code is not much more than a simple text explanation of the desired state. A user named *Joe User* should be *present*, a home directory for the user should be created, etc. It is very clear, very easy to read. Exactly how the user should be created is not within the code, nor are instructions for handling different operating systems.

Declarative language is much easier to read, and less prone to breakage due to environment differences. Puppet was designed to achieve consistent and repeatable results. You describe what the final state of the resource should be. Puppet will compare and implement any necessary changes to bring the resource into the desired state.

Conclusion

Conventional programming languages create change by listing exact operations which should be performed. Code which defines each procedure is known as Procedural Programming.

Good Puppet manifests are written using declarative programming. Instead of defining exactly how to make changes, in which you must write code to test and compare the system state before making that change, you instead declare how it should be. It is up to the Puppet agent to compare, evaluate, and implement the necessary changes.

Declarative programming is easier to create, easier to read, and easier to maintain.

CHAPTER 2

Creating a Learning Environment

In this chapter we will create a virtualized environment suitable for learning and testing Puppet. We will utilize Vagrant and Virtualbox to set up this environment on your desktop or laptop. You'll keep this environment long after you have finished this book.

If you are an experienced developer or operations engineer, you are welcome to use a testing environment of your own choice. Anything which can host multiple Linux nodes will work. Puppet's needs are minimal. Any of the following would be suitable for use as a Puppet test lab:

- A bunch of spare systems you have sitting around that you can install Linux on.
- An AWS Free Tier Amazon Web Services instance.
- An OpenStack DevStack development instance.
- An VMware Free vSphere ESXi solo instance.
- A Vagrant development environment on your personal computer.

You can build your own test lab using one of the solutions above, or you can use an existing test lab you maintain. In all cases I recommend using an OS compatible with RedHat Enterprise Linux 6 or 7 for learning purposes. The CentOS platform is freely available, and fully supported by both Red Hat and Puppet Labs. This will allow you to breeze through the learning exercises without distractions. After you have a working Puppet setup, you can detailed notes in the Appendix for usage with other operating systems.

We recommend and are going to use Vagrant for the remainder of this book, for the following reasons:

- It is easier for you to setup and get started quickly.
- You can more easily carry it with you, and restart it at any time.

- The Vagrant setup we provide gives you copies of the puppet manifest and configuration files used in this book.
- You can always build one of the other environments later for a comparison point.

If you plan to use your own testing environment, skip ahead to Initialize Non-Vagrant System near the end of this chapter.

If you are going to use Vagrant as we recommend, let's get started installing it. You'll need to download two packages.

Go to https://www.virtualbox.org/wiki/Downloads and download the appropriate platform package for your system.

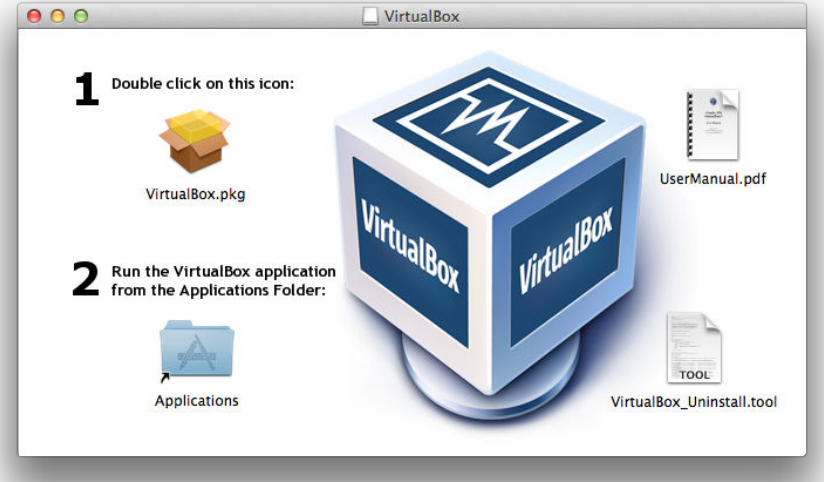
Next, go to http://www.vagrantup.com/downloads/ and download the appropriate platform package for your system.

You should install these packages according to the instructions for your operating system, below:

Installing Vagrant on Mac

First you should run the VirtualBox installer. Open to Virtualbox DMG image file you downloaded and click on the **Virtualbox.pkg** installer.

Figure 2-1. VirtualBox package installer for Mac



Accept the license and the installer will will complete the installation.

Next you should run the Vagrant installer. Open to Vagrant DMG image file you downloaded and click on the **Vagrant.pkg** installer.

Figure 2-2. Vagrant package installer for Mac

Likewise, accept the license agreement and the installer will complete the installation.

Finally, you will need to install Xcode on your Macintosh, if you don't have it already. Perform the following steps:

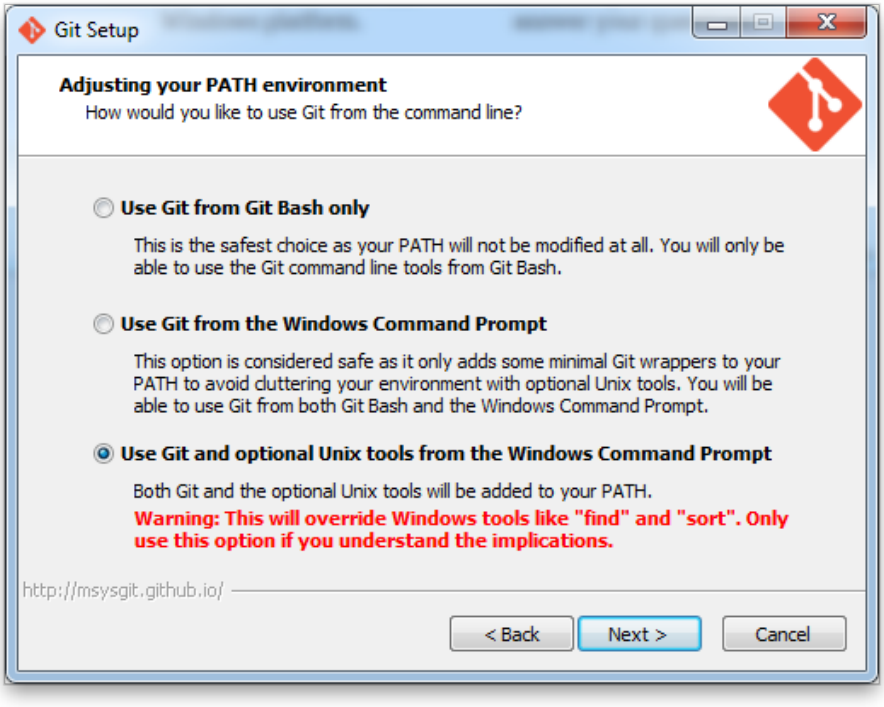
1. Pull down on the Apple logo at the top left of your screen.
2. Select **App Store...**
3. In the App Store, type "xcode" into the search bar on the right.
4. Click **Install** beneath the top left icon *Xcode: Developer Tools*

Installing Vagrant on Windows

First, install the Git client for windows with Unix tools.

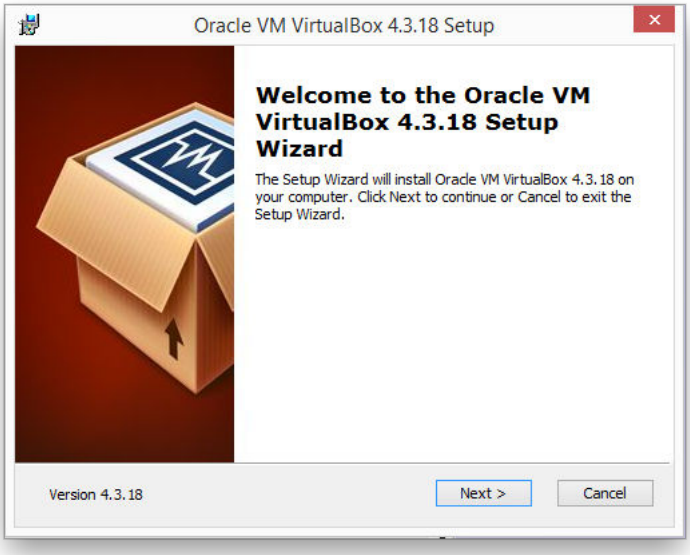
1. Browse to http://git-scm.com/ and download the Windows GUI client. (available directly at http://git-scm.com/download/win)
2. Run the installer you have downloaded.
3. Allow it to make changes to the local system.
4. Accept the default components.
5. On the Git Setup screen, select the option to **Use Git and optional Unix tools from the Windows Command Line**.
6. Finish the installation.

Figure 2-3. Git Setup for Windows



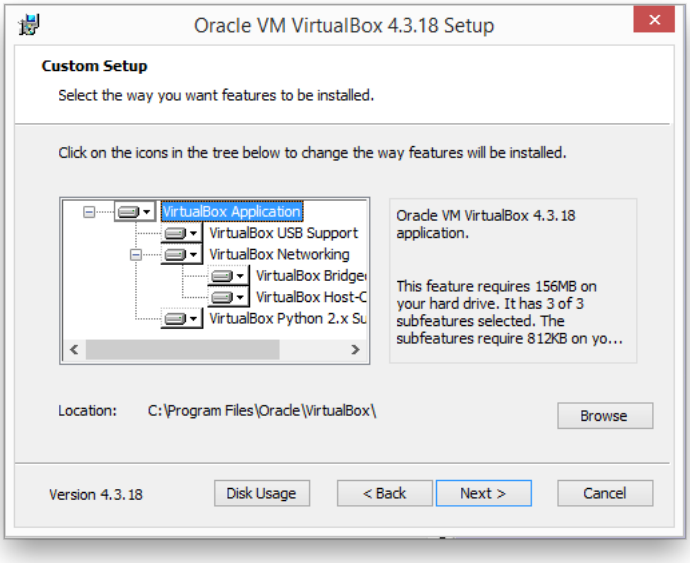
Installing VirtualBox is very straightforward. Run the VirtualBox installer package that you downloaded.

Figure 2-4. VirtualBox Installer for Windows



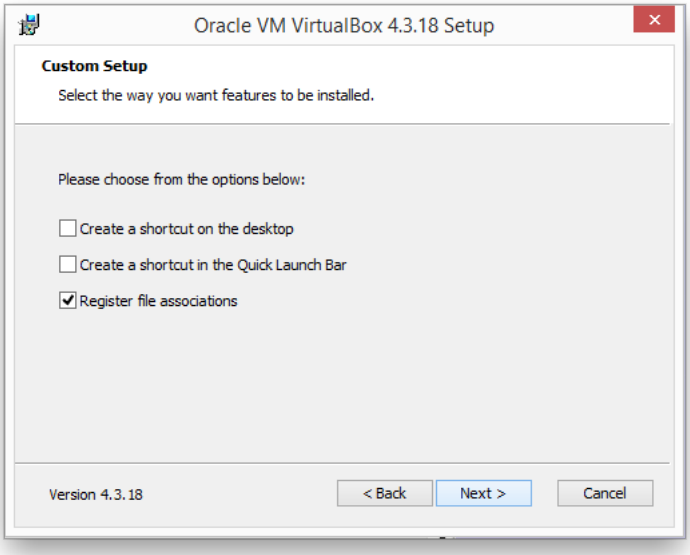
Click **Next** to install all features

Figure 2-5. Install all Features



I disable the options to create shortcuts, as these won't be necessary for the learning environment.

Figure 2-6. Register File Associations

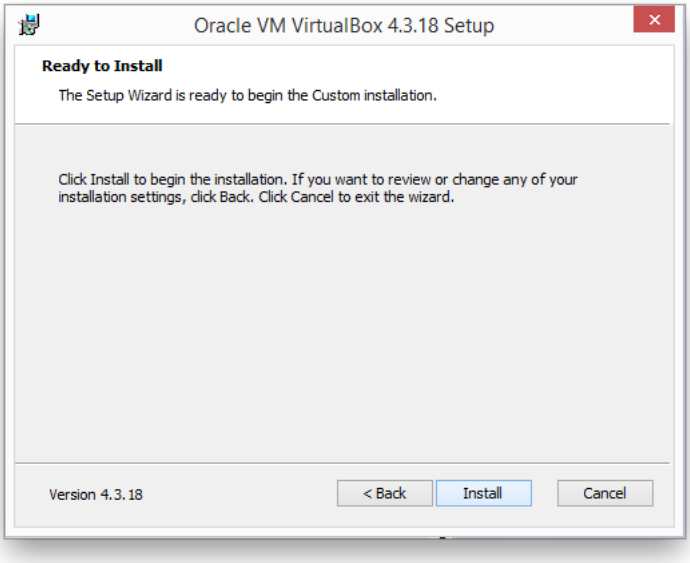


Accept the warning about a short interruption to your networking.

Figure 2-7. VirtualBox Network Interruption Notice

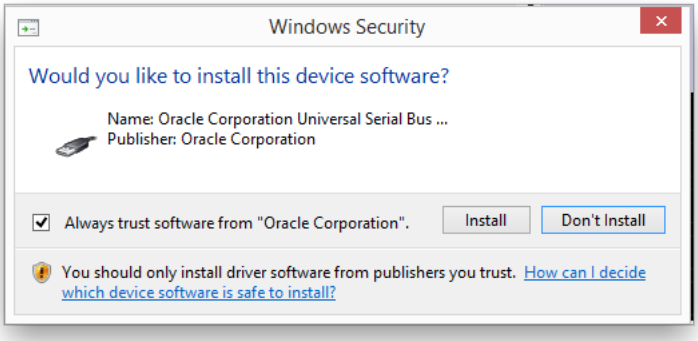
Click **Install** to install VirtualBox.

Figure 2-8. Install VirtualBox



You don't necessarily need to install the USB drivers for our environment, but it doesn't hurt anything.

Figure 2-9. USB Drivers

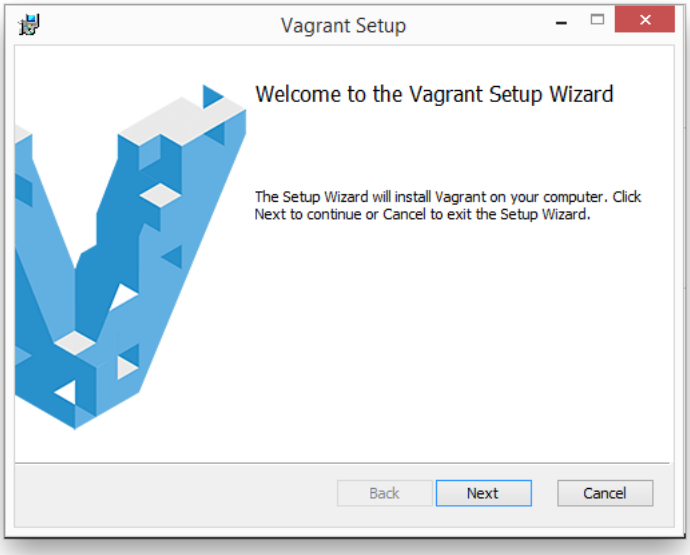


Disable the checkbox to start VirtualBox after installation. We'll start it later in the process.

Figure 2-10. Disable autostart

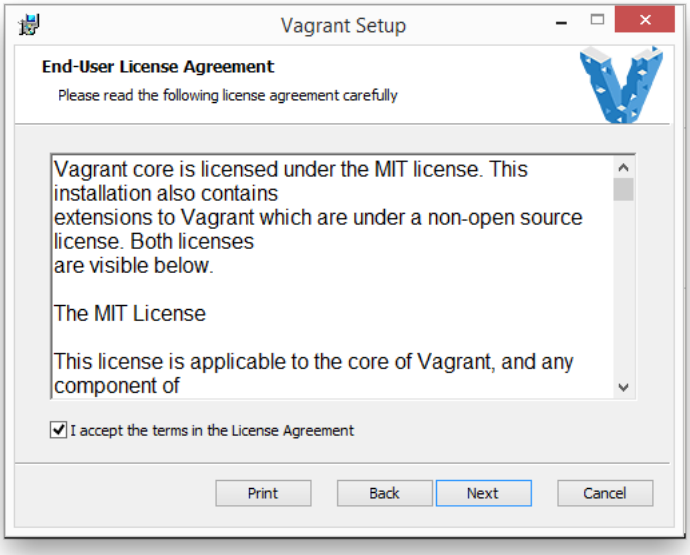
Now we should install Vagrant. Run the Vagrant installer package that you downloaded.

Figure 2-11. Vagrant Installer for Windows



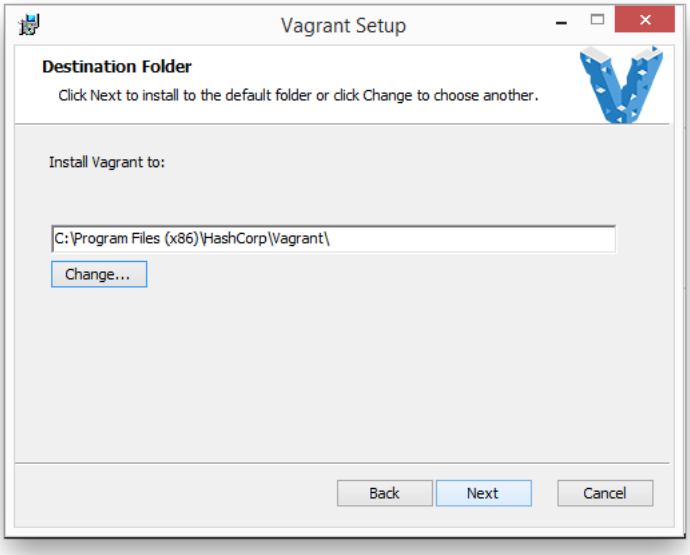
Accept the License Agreement

Figure 2-12. License Agreement



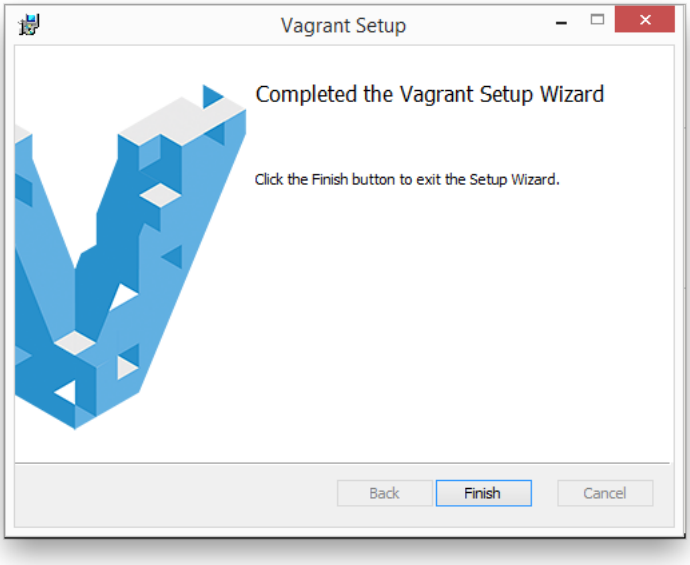
Select where you want to install Vagrant. As this figure shows, I prefer to install Vagrant in the normal system location for 64-bit programs. It doesn't matter which path you choose here.

Figure 2-13. Vagrant Destination Folder



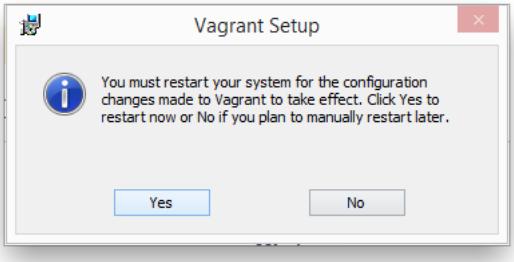
This acknowledged completion of the installation.

Figure 2-14. Installation Complete



Windows systems will need to reboot at this point.

Figure 2-15. Reboot after installation



Starting a Command Prompt

At this point we'll need to start a command prompt. We'll be using the command prompt for the remainder of this book, so now is a good time to get used to it.

On a Macintosh, follow these steps:

1. Open Finder
2. Click on **Applications** in the sidebar on the left.
3. Open the **Utilities** folder.
4. Start the **Terminal** application.

On Windows 7 and before, follow these steps:

1. Touch the Windows key on your keyboard, or click on the Start button on the bottom left of your screen.
2. Type **cmd** into the *Search programs and files* search bar immediately above the Start button.

On Windows 8 and later, follow these steps:

1. From the Metro screen, click on the down arrow icon at the bottom left of the screen.
2. Scroll to the right and locate the **Windows System** section.
3. Click on **Command Prompt**.

No matter which operating system you are using, you will find yourself at a command prompt in your home directory. This is where we will start.

If you are an experience user you may already have a terminal or command prompt that you prefer to use. Any alternative should work just fine.

Downloading a Box

Now we will download a virtual box image to use as the base system for our learning environment. As mentioned before, we'll use CentOS 7.0 as it is well supported by all Puppet Labs programs and modules.

```
$ vagrant box add --provider virtualbox \
    http://puppet-vagrant-boxes.puppetlabs.com/centos-65-x64-virtualbox-nocm.box
==> box: Loading metadata for box 'https://atlas.hashicorp.com/puppetlabs/boxes/centos-7.0-64-nocm
==> box: Adding box 'puppetlabs/centos-7.0-64-nocm' (v1.0.1) for provider: virtualbox
    box: Downloading: https://atlas.hashicorp.com/puppetlabs/boxes/centos-7.0-64-nocm/versions/1.0
==> box: Successfully added box 'puppetlabs/centos-7.0-64-nocm' (v1.0.1) for 'virtualbox'!
```

On windows this would look like this:

```
C:\> vagrant box add --provider virtualbox \
       https://atlas.hashicorp.com/puppetlabs/boxes/centos-7.0-64-nocm
==> box: Loading metadata for box 'https://atlas.hashicorp.com/puppetlabs/boxes/centos-7.0-64-nocm
==> box: Adding box 'puppetlabs/centos-7.0-64-nocm' (v1.0.1) for provider: virtualbox
    box: Downloading: https://atlas.hashicorp.com/puppetlabs/boxes/centos-7.0-64-nocm/versions/1.0
==> box: Successfully added box 'puppetlabs/centos-7.0-64-nocm' (v1.0.1) for 'virtualbox'!
```

You may note that there are boxes at this site that already have Puppet installed. I want you to go through the installation of Puppet on a barebones system so that you have that experience before we get started.

Initialize Vagrant System

Now we shall create a project directory for your learning environment. Starting in your home directory, take the following steps. Steps are the same for Windows or Macintosh users, except that Windows users will have a `C:\>` prompt.

```
$ git clone https://github.com/jorhett/learning-puppet4
Cloning into 'learning-puppet4'...
remote: Counting objects: 64, done.
remote: Total 64 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (64/64), done.
Checking connectivity... done.
$ cd learning-puppet4
```

You'll notice that we have preinstalled a *Vagrantfile* which lists the systems we'll use in this book. If you are familiar with Vagrant, you'll know that we can easily start

these systems with `vagrant up`. Let's do that now. This will initialize a client system we'll use for learning puppet.

```
$ vagrant up client
Bringing machine 'client' up with 'virtualbox' provider...
==> client: Importing base box 'centos65'...
==> client: Matching MAC address for NAT networking...
==> client: Setting the name of the VM: puppet-intro_client_1415082034018_51797
==> client: Clearing any previously set network interfaces...
==> client: Preparing network interfaces based on configuration...
    client: Adapter 1: nat
    client: Adapter 2: hostonly
==> client: Forwarding ports...
    client: 22 => 2222 (adapter 1)
==> client: Booting VM...
==> client: Waiting for machine to boot. This may take a few minutes...
    client: SSH address: 127.0.0.1:2222
    client: SSH username: vagrant
    client: SSH auth method: private key
    client: Warning: Connection timeout. Retrying...
==> client: Machine booted and ready!
==> client: Checking for guest additions in VM...
==> client: Setting hostname...
==> client: Configuring and enabling network interfaces...
==> client: Mounting shared folders...
    client: /vagrant => /Users/jorhett/puppet-intro
```

We've started only a single machine to learn from, named *client*. There are several other machines available that we will utilize in future chapters.

```
$ vagrant status
Current machine states:

client                    running (virtualbox)
puppetmaster              not created (virtualbox)
web1                      not created (virtualbox)
web2                      not created (virtualbox)
web3                      not created (virtualbox)
dashboard                 not created (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

You can suspend, resume, and destroy these instances quite easily.

```
$ vagrant suspend client
==> client: Saving VM state and suspending execution...

$ vagrant resume client
==> client: Resuming suspended VM...
==> client: Booting VM...
```

```
==> client: Waiting for machine to boot. This may take a few minutes...
    client: SSH address: 127.0.0.1:2222
    client: SSH username: vagrant
    client: SSH auth method: private key
    client: Warning: Connection refused. Retrying...
==> client: Machine booted and ready!

$ vagrant destroy client
    client: Are you sure you want to destroy the 'client' VM? [y/N] n
==> client: The VM 'client' will not be destroyed, since the confirmation
==> client: was declined.
```

If you do destroy the instance, all you need to do is `vagrant up client` to create another and start over. We're not going to spend any more time covering Vagrant here, but remember `suspend` and `resume` when you want to stop your test environment but don't want to have to start from scratch when you restart it later.

Now that this is running, let's login to the client system and get started.

```
$ vagrant ssh client
Last login: Tue Feb 18 13:49:31 2014 from 10.0.2.2
Welcome to your Packer-built virtual machine.
[vagrant@client ~]$
```

Initialize Non-Vagrant System

If you are using a virtual node of your own choice, we'll need to take a couple of steps so that you can follow the instructions in this book. Login to the virtual node and run the following commands:

```
$ git clone https://github.com/jorhett/learning-puppet4
Cloning into 'learning-puppet4'...
remote: Counting objects: 64, done.
remote: Total 64 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (64/64), done.
Checking connectivity... done.
$ sudo ln -s /home/username/learning-puppet4 /vagrant
```

Within this book you will many times see the following prompt shown in the examples:

```
[vagrant@client ~]$
```

You'll need to mentally replace this with whatever your virtual node's shell prompt is. Or if you wish to be ensure it looks like same, you can use the following environment variable to sync them up:

```
$ export PS1='[vagrant@client \W]\$ '
```

Choosing a Text Editor

Before we go on to install Puppet on your virtual system, stop and take a moment to ensure you have a text editor you like handy and available. There are two ways to use text editors.

Inside the virtual system
: If you are comfortable using unix text editors like **vim**, **emacs**, or **nano** you can run your editor inside the virtual system.

On your desktop
: The folder `learning-puppet4` is mounted inside your virtual system as `/vagrant`. This means you can use a text editor of your choice to edit files inside this directory, and they will be visible and available to Puppet on your virtual system.

One issue for both Mac and Windows users is handling of carriage returns and linefeeds within files. Puppet and Git both work best when lines are terminated by linefeeds without carriage returns. This is well known as "Unix file format." So it is important to select an editor which can read files in this format, and perhaps even more importantly, will write them out in the same format without any surprises.

Many editors on Windows open Unix format files correctly, but will quietly replace the endings with a carriage return when saving. As your target nodes may expect linefeed-terminated files, this can cause file corruption on the target host. Likewise, the opposite problem can exist as well: if you are editing files to be written out to Windows nodes, then you may need to preserve the carriage returns in the file. If you have a mixed environment of Unix/Linux and Windows nodes, it is essential that your editor can handle both file formats.

Here are a list of editors we recommend. I have limited this list to only include editors which handle both file formats well. If you don't have any Windows nodes, then you can use any editor that pleases you.

On the Virtual System

There are three editors immediately available to you on the virtual system. All three of these normally create Unix format files, but can edit Windows format files without causing corruption. You can use the following editors from the command line inside your virtual system.

Table 2-1. Text editors available on the virtual system

vim	Install with `sudo yum install vim`	Powerful editor for experienced users, see Learning the vi and Vim Editors

emacs	Install with `sudo yum install emacs-nox`	Powerful editor for experienced users, see Learning GNU Emacs
nano	Install with `sudo yum install nano`	An easy text editor for beginners, no book required!

As it happens, all three of the above editors are installed on Macs by default, and available with the Cygwin package for Windows. You can get quite comfortable using these editors on your desktop.

If you use VIM, I highly recommend that you download and install the Puppet syntax highlighter, available at https://github.com/rodjek/vim-puppet

On your Desktop

When writing Puppet modules using a Windows system, you'll run into problems with line-endings within templates and source files. Windows uses both a carriage-return character and a linefeed character to end a line, whereas Mac, Linux, and Unix systems use only the linefeed character.

If you open up the files we use in this book with the Windows Notepad editor, you will see that they show up as a single, unbroken line. Wordpad can display the file but will change the endings when it writes out changes. None of the built-in editors on Windows: Notepad, Wordpad, or Word are safe to use with Unix format files.

For Windows users I highly recommend the Notepad++ editor. It can open and write out both Unix and Windows format files without changing the line endings. It does not reformat files without explicit action taken by you.

For Mac users I recommend the TextWrangler and TextMate editors. The built-in TextEdit editor is minimally sufficient for Unix format files, but cannot handle files in Windows format properly.

If you are already a fan of the Unix editor Vim, you can find a GUI version of it for your operating system at GVim or MacVIM. Vim can safely read and write files in both formats.

If you have experience with the Eclipse IDE, the workspace editor can safely read, write, and convert upon request files in both Unix and Windows formats. Later on in this book I'll show you how to install Geppetto, an Eclipse extension that will help you develop and debug Puppet modules and manifests within Eclipse.

Conclusion

In this chapter you created a virtualized environment suitable for learning and testing Puppet without affecting any production or personal systems.

The default learning environment we recommended had you install the Git development tools, Vagrant, and Virtualbox to provide a virtualized CentOS system on which you can test and develop.

If you already have a virtualization platform that you prefer to work with, you are welcome to use that instead. We did recommend using an operating system compatible with RedHat Enterprise Linux 6 or 7, as it is the best tested and guaranteed version for compatibility with Puppet Labs-provided modules that we will be using through the book. After you feel strong in the use of Puppet, there are detailed notes for use on other platforms in the Appendices.

You cloned a Git repository which contains Vagrant system definitions, some example manifests, and other helpful files we'll use throughout this book.

Finally, we discussed the need for editing text files, and how you can use either an editor on the virtualized system, or an editor on your desktop to create and edit Puppet manifests.

This environment is yours to keep. It will be useful as you develop and test Puppet manifests and modules during your learning process.

CHAPTER 3

Installing Puppet

In this chapter you will install the Puppet agent and its dependencies. We have deliberately chosen a Vagrant box that doesn't have Puppet pre-installed, so that you can go through and learn the process. You can repeat these exact steps on any test or production node to install or upgrade Puppet.

Let's get started by installing the Puppet Labs package repository.

Adding the Package Repository

Now we shall install and enable the Puppet Labs package collection 1 repository on your fresh new system. Starting in the home directory, take the following steps.

```
[vagrant@client ~]$ sudo yum install -y http://yum.puppetlabs.com/puppetlabs-release-pc1-el-7.noar
```

This command will install and enable the Puppet Labs *puppetlabs-release-pc1* package repository, which contains the Puppet 4 package. After it has finished installing, you can confirm it is enabled with the following command.

```
[vagrant@client ~]$ sudo yum repolist
Loaded plugins: fastestmirror
...snip repository checks...
repo id                      repo name                                    status
base/7/x86_64                CentOS-7 - Base                              8,652
extras/7/x86_64              CentOS-7 - Extras                               84
pl-puppet-agent-latest       PL Repo for puppet-agent at commit latest        1
puppetlabs-pc1/x86_64        Puppet Labs PC1 Repository el 7 - x86_64         6
updates/7/x86_64             CentOS-7 - Updates                             355
repolist: 9,037
```

This shows you that there was, at the time this book was last updated, 6 packages in the *puppetlabs-pc1* repository.

What is a Package Collection?

The Puppet ecosystem contains many tightly related and dependent packages. Puppet, Facter, MCollective, and the Ruby interpreter are all tightly related dependencies. The Puppet agent, Puppet server, and PuppetDB are self-standing but interdependent applications.

Production Puppet environments have been struggling with two conflicting needs:

- It is important to stay up to date with the latest improvements and security fixes.
- Improvements and upgrades in an application would sometimes introduce problems for interdependent components of the Puppet ecosystem.

Puppet Labs has chosen to address these concerns with two related changes.

Puppet and all core dependencies are shipped together in a single package.
: This change reduces the need to ensure compatibility across a wide variety of versions of dependencies. It also ensures that modern versions of Ruby are available on every supported operating system.

Components of the Puppet ecosystem will be tested, packaged, and shipped together in Package Collections.
: Significant improvements and breaking changes will be introduced in a new Package Collection. This allows Puppet environments to safely track updates within a Package Collection, knowing that all versions within the collection are tested and guaranteed to work together.

We'll cover how to manage upgrades of the Package Collection in Part IV of this book.

Installing the Puppet Agent

Now let's go ahead and install the Puppet Agent.

```
[vagrant@client ~]$ sudo yum install -y puppet-agent
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
 * base: centos.sonn.com
 * extras: mirrors.loosefoot.com
 * updates: mirrors.sonic.net
Resolving Dependencies
--> Running transaction check
---> Package puppet-agent.x86_64 0:1.1.0-1.el7 will be installed
--> Finished Dependency Resolution
```

```
...snip lots of output...

Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Installing : puppet-agent-1.1.0-1.el7.x86_64             1/1
  Verifying  : puppet-agent-1.1.0-1.el7.x86_64             1/1

Installed:
  puppet-agent.x86_64 0:1.1.0-1.el7

Complete!
```

Reviewing Dependencies

If you have installed previous versions of Puppet, you were used to seeing dependency packages installed along with Puppet. Puppet 4 uses an All In One (AIO) installer, where all dependencies are installed together with Puppet. You can view these in the new installation directory.

```
[vagrant@client ~]$ ls -la /opt/puppetlabs/bin/
total 0
drwxr-xr-x 2 root root 68 Apr  6 04:41 .
drwxr-xr-x 4 root root 29 Apr  6 04:41 ..
lrwxrwxrwx 1 root root 34 Apr  6 04:41 cfacter -> /opt/puppetlabs/puppet/bin/cfacter
lrwxrwxrwx 1 root root 33 Apr  6 04:41 facter -> /opt/puppetlabs/puppet/bin/facter
lrwxrwxrwx 1 root root 32 Apr  6 04:41 hiera -> /opt/puppetlabs/puppet/bin/hiera
lrwxrwxrwx 1 root root 30 Apr  6 04:41 mco -> /opt/puppetlabs/puppet/bin/mco
lrwxrwxrwx 1 root root 33 Apr  6 04:41 puppet -> /opt/puppetlabs/puppet/bin/puppet
```

Unlike previous versions of Puppet, the Puppet user commands are not installed in */usr/bin*, and won't be available in your path. There are several ways to deal with this:

- You can symbolically link the Puppet commands into the expected path.
- The */opt/puppetlabs/bin* directory can be added to your path, and to the `secure_path` in your *sudoers* file.

For this book, set up the symbolic links for simplicity:

```
[vagrant@client ~]$ ln -s /opt/puppetlabs/bin/puppet /usr/bin/puppet
[vagrant@client ~]$ ln -s /opt/puppetlabs/bin/facter /usr/bin/facter
[vagrant@client ~]$ ln -s /opt/puppetlabs/bin/mco /usr/bin/mco
```

Let's review the other programs you see in this directory besides Puppet.

Facter
: Facter is a program which evaluates a system and provides a number of *facts* about it. These facts include node-specific information like architecture, hostname, and IP address, in addition to custom information from plugins provided by Puppet modules. For a sneak preview, run the command `facter` right now and look at all the information it has.

  We'll be covering how to make use of `facter` facts in Using Puppet Configuration Language, and how to create custom facts in Part II: Puppet Modules.

Hiera
: Hiera is a component we'll use to load in the data used by Puppet manifests and modules. Hiera provides a configurable hierarchy allowing you to provide default values, and then override or expand them through a customizable hierarchy. This may sound complex, but Hiera's beauty and elegance comes from its simplicity.

  You'll learn how to use Hiera in Part II: Puppet Modules, after which you'll see Hiera used in every following example throughout this book.

Mco
: Mco is the command line client for the Marionette Collective, an orchestration tool tightly integrated with Puppet.

  You'll learn how to use MCollective in Part IV: Advanced Puppet, where we'll use the `mco` client to manipulate the Puppet agent.

Reviewing Puppet4 Changes

If you have installed previous versions of Puppet, the installation packages and the configuration file paths have all changed.

If you are new to Puppet, you can skim lightly through this section as we'll bring up each path again as we teach you how to use Puppet.

Linux and Unix

For Unix and Linux systems, the following changes have taken place.

Puppet is now installed using a new All-in-One (AIO) puppet-agent package.
: This AIO package includes private versions of Facter, Hiera, MCollective, and Ruby.

Executables are in /opt/puppetlabs/bin/
: All executables have been moved to */opt/puppetlabs/puppet/bin*. Executibles which should be in your path have symlinks in */opt/puppetlabs/bin*. You'll need to add this directory to your path, or symlink the executibles somewhere else in your path.

A private copy of Ruby 2.1.5 is installed in /opt/puppetlabs/puppet
: Ruby and supporting commands like `gem` are installed in */opt/puppetlabs/puppet*, to avoid them being accidentally called by users.

The configuration directory `$confdir` *is now /etc/puppetlabs/puppet*
: Open source Puppet now uses the same configuration directory as Puppet Enterprise. Files in */etc/puppet* will be ignored.

`$ssldir` *is inside* `$confdir`
: On most platforms Puppet put SSL keys and certificates in */var/lib/puppet/ssl*. With Puppet4 all SSL files will always be installed inside the `$confdir`.

MCollective configuration directory is now `/etc/puppetlabs/mcollective`
: Files in */etc/mcollective* will be ignored.

`$vardir` *for Puppet agent/apply is now /opt/puppetlabs/puppet/cache/*
: This new directory is used only by the Puppet agent and Puppet apply. At the time of this update, I cannot find a configuration variable to change this directory. and is configured in `$confdir`.

`$rundir` *for Puppet agent is now /var/run/puppetlabs*
: This directory stores PID files only, and can be changed in `$confdir`.

Modules, manifests, and the hiera config file have a new directory: /etc/puppetlabs/code
: Files which configure nodes have moved from `$confdir` to a new directory `$codedir`. This directory contains:

  - The *environments* directory for `$environmentpath`
  - The *modules* directory for `$basemodulepath`
  - The *hiera.yaml* config file for `$hiera_config`
  - The *hieradata* directory (specified in hiera config file)

The author feels strongly that placing storage of SSL certificates within the /etc directory is a violation of the Linux Filesystem Hierarchy Standard. The /etc directory should contain only static configuration files. In virtualized, auto-scaled environments new Puppet clients come and go. Some of my environments would build and destroy 50,000 nodes in a single day. This makes the SSL directory highly volatile, and completely unsuitable for placement within */etc/*. I have opened Bug PUP-4376 concerning this obvious mistake.

Windows

On Windows, very little has changed. The Puppet package has always been an All-In-One (AIO) installer. The package now includes MCollective. Executables remain in the same location, and the MSI package still adds Puppet's tools to the PATH. The `$confdir` and `$rundir` have not changed.

For review, the file locations are:

`$confdir`
: `COMMON_APPDATA`: defaults to C:\ProgramData\PuppetLabs\puppet\etc on modern Windows versions.

`$codedir`
: C:\ProgramData\PuppetLabs\code

`$vardir`
: C:\ProgramData\PuppetLabs\puppet\cache

`$rundir`
: C:\ProgramData\PuppetLabs\puppet\var\run

Making Tests Convenient

Throughout this book we'll be having you edit files and make changes within the Puppet configuration directory. Normally this would require you to type `sudo` every time you want to change one of those files.

If you're a stickler like I am, you may want to skip this next step and type `sudo` when modifying files in the puppet configuration directory. However I have found that most people find the following change to create an easier to use environment for learning. This command will make the vagrant user the owner of all files in that directory.

```
[vagrant@client ~]$ sudo chown -R vagrant /etc/puppetlabs
```

Obviously you won't be doing this in real production environment. However it is not uncommon to see something like the following done instead to give write access to all members of a certain group.

```
[vagrant@client ~]$ sudo chgrp -R sysadmin /etc/puppetlabs
[vagrant@client ~]$ sudo chmod -R g+w /etc/puppetlabs
```

Conclusion

In this chapter you

1. learned how to enable the Puppet Yum repository on your system.
2. installed Puppet and its dependencies from this repository.
3. reviewed the supporting dependencies that support Puppet.
4. reviewed the changed paths used by Puppet 4 on Unix/Linux systems.

Best of all, none of this was done with magic helper scripts that hid the details from you. You can repeat these exact steps on any test or production node to install or upgrade Puppet.

CHAPTER 4

Writing Manifests

The very first concept we want to introduce you to is the Puppet **manifest**. A manifest is a file containing Puppet configuration language that describes how resources should be configured. The manifest is the closest thing to what one might consider a Puppet *program*. It uses resources to define a policy to be enforced on a node. It is therefore the base component for Puppet configuration policy, and a building block for complex Puppet modules.

This chapter will focus on how to write configuration policies for Puppet 4 manifests. Writing manifests well is the single most important part of building Puppet policies.

Let's get started with the smallest component within a manifest.

Implementing Resources

Resources are the smallest building block of the Puppet configuration language. They represent a singular element which you wish to evaluate, create, or remove. Puppet comes with many built-in resources. These default resources manipulate system components that you are already familiar with, including

- Users
- Groups
- Files
- Hosts
- Packages
- Services

...and many more. Furthermore, you can create your own resources. However, let's get started with one of the simplest resources--the Notify resource. Let's start with the stereotypical first program written in every language.

```
notify { 'greeting':
  message => 'Hello, world!'
}
```

This code declares a notify resource named *greeting*. It has a single attribute, `message` which has the value we'd expect in our first program. Attributes are separated from their values using a *fat comma* (also called a *hash rocket*), which is a very common way to identify key/value pairs in Perl, Ruby, and PHP scripting languages.

This tiny bit of code is a fully functional and valid manifest. This manifest (with one single resource) does only one thing, which is to output that stereotypical message every time it is called. Let's go ahead and use Puppet to evaluate this manifest.

As part of the definition of your virtual system, we have preinstalled some Puppet manifests in the `/vagrant/manifests` directory for use in this class. We'll refer to these throughout the book.

```
[vagrant@client ~]$ cat /vagrant/manifests/helloworld.pp

notify { 'greeting':
  message => 'Hello, world!'
}
```

As you can see from this example, manifests are text files named with a `.pp` file extension which describe resources using the Puppet configuration language. You can create or modify a Puppet manifest using any text editor.

Applying a Manifest

One of the best features of Puppet is the ease of testing your code. Puppet does not require you to set up complicated testing environments to evaluate Puppet manifests. It is easy, nay, downright trivial to test a Puppet manifest.

Let's go ahead and implement this manifest. We will do this using the `puppet apply` command which tells Puppet to implement a single Puppet manifest.

```
[vagrant@client ~]$ puppet apply /vagrant/manifests/helloworld.pp
Notice: Compiled catalog for client.example.com in environment production in 0.02 seconds
Notice: Hello, world!
Notice: /Stage[main]/Main/Notify[greeting]/message: defined 'message' as 'Hello, world!'
Notice: Finished catalog run in 0.01 seconds
```

As you can see, Puppet has implemented the manifest. It does this in several steps:

1. Compiles the manifest into a *Puppet catalog*.

2. Uses dependency information (covered soon) to determine which resources should be handled first.
3. Evaluates the resource to determine if changes are necessary.
4. Creates, modifies, or removes the resource--a notification message is created.
5. Provides verbose feedback about the catalog implementation.

Don't worry about memorizing these steps at this point in the learning process. We simply wanted to introduce you to the ideas here, which we will be covering in increasing depth throughout this book. We'll cover the catalog compilation and evaluation process in great detail at the end of Part I.

For now, just remember that you'll use `puppet apply` to implement Puppet manifests. It will provide you verbose feedback on actions Puppet took to bring the target resources into alignment with the declared policy.

Declaring Resources

There are only a few rules to remember when declaring resources. The format is always the same:

```
resource_type { 'resource_title':
  ensure     => present,         # usually 'present' or 'absent'
  attribute1 => 1234,            # number
  attribute2 => 'value',         # string
  attribute3 => ['red','blue'],  # array
  noop       => false,           # boolean
}
```

Don't get hung up analyzing the data types shown in this example. We will cover the different data types of Puppet 4 exhaustively in Using Puppet Configuration Language.

The most important rule for resources is: **There can be only one**. Within a manifest or set of manifests being applied together (the *catalog* for a node) a resource of a given type can only be declared with a given title once. Any other resource of that type should refer to a different underlying component, and have a unique title.

For example, the following manifest will fail because the same title is used for both file resources.

```
[vagrant@client ~]$ cat myfile.pp
file { 'my_file':
  ensure => present,
  path   => 'my_file.txt',
}
```

```
file { 'my_file':
  ensure => present,
  path   => 'my_file.csv',
}

notice { 'my_file':
  message => "My file is present",
}

[vagrant@client ~]$ puppet apply myfile.pp
Error: Evaluation Error: Error while evaluating a Resource Statement, Duplicate declaration: File[
```

You'll notice that no complaint was given for the Notice resource with the same title. This is not a conflict. Only resources of the same type cannot utilize the same title. Naming the files above with their full paths ensures no conflicts:

```
file { '/home/vagrant/my_file.txt':
  ensure => present,
  path   => '/home/vagrant/my_file.txt',
}

file { '/home/vagrant/my_file.csv':
  ensure => present,
  path   => '/home/vagrant/my_file.csv',
}
```

Viewing Resources

One nice feature of Puppet is that it can show you an existing resource written out in Puppet language. This makes it easy to generate code based on existing configurations. Let's demonstrate this with an e-mail alias.

```
[vagrant@client ~]$ puppet resource mailalias postmaster
mailalias { 'postmaster':
  ensure    => 'present',
  recipient => ['root'],
  target    => '/etc/aliases',
}
```

This output gives you the structure, syntax, and attributes to declare this alias within your Puppet policies. You could write this output to a manifest file, change the recipient, and then use `puppet apply` on this manifest to change the postmaster alias on this node.

Let's examine another resource--the user you are logged in as.

```
[vagrant@client ~]$ puppet resource user vagrant
Error: Could not run: undefined method `exists?' for nil:NilClass
```

This somewhat confusing error message means that you don't have the privileges to view that resource. So let's escalate our privileges to complete this command with `sudo`.

```
[vagrant@client ~]$ sudo puppet resource user vagrant
user { 'vagrant':
  ensure           => 'present',
  gid              => '500',
  groups           => ['wheel'],
  home             => '/home/vagrant',
  password         => '$1$sC3NqLSG$FsXVyW7azpoh76edOfAWm1',
  password_max_age => '99999',
  password_min_age => '0',
  shell            => '/bin/bash',
  uid              => '500',
}
```

If you look at the resource you'll see why root access was necesarry. The user resource contains the user's password hash, which required root privilege to read from the shadow file. As with the alias above, you could write this to a file, replace the password hash, and use `sudo puppet apply` to change the root password.

Executing Programs

Let's examine another resource types: commands, or execs. You can use an `exec` resource to execute programs as part of your manifest. Let's examine of these now.

```
exec { 'echo-holy-cow':
  path      => ['/bin'],
  cwd       => '/tmp',
  command   => 'echo "holy cow!" > testfile.txt',
  creates   => '/tmp/testfile.txt',
  returns   => [0],
  logoutput => on_failure,
}
```

Now when you apply this manifest, it will create the *testfile.txt* file. Notice that we use single quotes to encapsulate the values given to the attributes.

Best Practice: It is recommended within the Puppet Style Guide to use single quotes for any value which does not contain a variable. This protects you against accidental interpolation of a variable that was not intended. Use double quotes with strings containing variables.

The `exec` resource defined above uses the `creates` attribute. This attribute defines what the expected result of the command execution will be. When the file named

exists, the command is not executed. This means the manifest can be run repeatedly and nothing will change after the file is initially created. Let's test this out here:

```
[vagrant@client ~]$ puppet apply /vagrant/manifests/tmp-testfile.pp
Notice: Compiled catalog for client.example.com in environment production in 0.03 seconds
Notice: /Stage[main]/Main/Exec[echo-holy-cow]/returns: executed successfully
Notice: Finished catalog run in 0.07 seconds

[vagrant@client ~]$ puppet apply /vagrant/manifests/tmp-testfile.pp
Notice: Compiled catalog for client.example.com in environment production in 0.03 seconds
Notice: Finished catalog run in 0.01 seconds
```

There are a wide variety of attributes you can use to control whether or not an exec resource will be execute, and which return codes indicate success or failure. This is a complex and feature-rich resource. Whenever implementing an `exec` it's best to test carefully, and refer to the documentation https://docs.puppetlabs.com/references/latest/type.html#exec.

I must beg your forgiveness, as I have deliberately led you astray to teach you a common mistake when learning declarative programming. While `exec` is an essential and sometimes crucial resource type, it is best to avoid using execs whenever possible. We'll implement the exact same result with a more appropriate File resource below.

If you examine the `exec` resource above, you'll note that we had to declare *how* to make the change, and also *whether* or not to make the change. This is very similar to procedural programming, and can be very difficult to maintain.

It is generally more difficult to write declarative code using an exec. One tends to fall backwards into a procedural programming style. Except for circumstances where no other method is possible, an exec is generally indication of a poorly written policy.

Best Practice: Avoid using `exec` whenever possible, especially when a native Puppet resource can do the job.

Managing Files

How else could we create this file? We could have used the `file` resource. Let's examine one now.

```
file { '/tmp/testfile.txt':
  ensure   => present,
  mode     => '0644',
```

```
  replace  => true,
  contents => 'holy cow!',
}
```

This is a properly declarative policy. We declare that the file should exist, and what the contents of the file should be. We do not need to concern ourselves with how, or when to make changes to the file. Furthermore, we were able to ensure the contents of the file remained consistent, which is not possible within an echo command.

You'll also notice that the declarative manifest used less lines of text, and guaranteed a more consistent output.

Let's go ahead and apply this policy now.

```
[vagrant@client ~]$ puppet apply /vagrant/manifests/file-testfile.pp
Notice: Compiled catalog for client.example.com in environment production in 0.06 seconds
Notice: /Stage[main]/Main/File[/tmp/testfile.txt]/content: content changed
   '{md5}0eb429526e5e170cd9ed4f84c24e442b' to '{md5}3d508c856685853ed8a168a290dd709c'
Notice: /Stage[main]/Main/File[/tmp/testfile.txt]/mode: mode changed '0664' to '0644'
Notice: Finished catalog run in 0.03 seconds

[vagrant@client ~]$ puppet apply /vagrant/manifests/file-testfile.pp
Notice: Compiled catalog for client.example.com in environment production in 0.07 seconds
Notice: Finished catalog run in 0.02 seconds
```

Unlike the previous `exec` resource, Puppet observed that the contents were different and changed the file to match. Now, you're probably thinking: aren't the file contents the same in both? Nope. It's not obvious in the exec declaration but `echo` appends a trailing newline to the text. As you can see here, the file contents don't include a newline.

```
[vagrant@client ~]$ cat /tmp/testfile.txt
holy cow![vagrant@client ~]$
```

You can easily adjust the file contents to include as many newline characters as you want. Since the newline character is interpreted, you'll need to use double quotes around the contents. You could also change the `replace` attribute if you only wanted to create the file but not replace it.

```
file { '/tmp/testfile.txt':
  ensure   => present,
  mode     => '0644',
  replace  => false,
  contents => "holy cow!\n",
}
```

You can find complete details of the many attributes available for the file resource at https://docs.puppetlabs.com/references/latest/type.html#file

Declarative Review

In the previous section I told you that it was best practice to avoid using Exec resources. To understand the reason for this, let's return our previous discussion of the word **Declarative**.

If the code informs the interpreter what to do, and when to do it, and how to do it... then it is functioning in a procedural way. If the code informs the interpreter what it wants the result to be, then the code is declarative.

It is unfortunately common to see people new to Puppet spend a lot of time attempting to write procedural manifests. They struggle with attempting to instruction Puppet on exactly how to do something. Eventually, they realize they are fighting against the nature of a declarative interpreter, and come around in their way of thinking. If you can learn this properly now, you can avoid the most common mistakes and move directly forward to becoming a Puppet expert.

Yes, the Exec resource will let you write procedural manifests in Puppet. However, just like we discussed in Thinking Declarative you'll find that it takes more effort, is harder to maintain, and is less likely to produce consistent results.

The reason I advocate so strongly to not use Execs is fairly simple. The Exec resource runs the command, but it doesn't really know what the exec-ed command does. That command could modify the node state in any form, and Puppet wouldn't be aware of it. Exec represents a *fire and forget* mentality, where the only piece of information returned to Puppet is the exit code from the command.

```
Notice: /Stage[main]/Main/Exec[echo-holy-cow]/returns: executed successfully
```

All you know is that the command here returned an exit code you expected (0 by default, which is the Unix convention for success). You don't really know what the command did. If someone wrote a Puppet manifest last week, and then removed it from the execution this week, you'll have to go search backups to determine what the command did.

When you declare the desired state using resources specific to what has been changed, then Puppet logs each and every change made. Files which are changed are backed up, and can be restored if necessary. For example, let's examine what happened when we implemented the File resource.

```
Notice: /Stage[main]/Main/File[/tmp/testfile.txt]/content: content changed
   '{md5}0eb429526e5e170cd9ed4f84c24e442b' to '{md5}3d508c856685853ed8a168a290dd709c'
Notice: /Stage[main]/Main/File[/tmp/testfile.txt]/mode: mode changed '0664' to '0644'
```

In this implementation, without having any access to read the original Puppet manifest, you know that the file mode and the file contents were changed. You can validate the file contents later to match the md5 hash, or restore a copy of the file as it was before this change was made.

Using the File resource is less code, easier to read, more consistent in implementation, and best of all logs much more accurately the changes which were made. I find this statement to be true of every situation when you can replace an Exec resource with a native resource. Use an Exec resource only when no other choice is available.

Testing Yourself

Let's pause for a second and utilize what you have learned to build a Puppet manifest.

- Use `puppet resource` to create a new manifest from the file */tmp/testfile.txt*.
- Change the file `mode` to be group writeable. (either *0664* or *ug=rw,o=r*)
- Change the content to say something that amuses you.
- Run **puppet apply *yourmanifest.pp*** and observe the changes.
- Confirm the changes with the **ls -la /tmp** and **cat /tmp/testfile.txt** commands.
- Run **puppet apply** again and see what happens.

You won't need to use sudo to make any of these changes, as you are the owner of this file.

Conclusion

In this chapter you learned about Manifests, single files which provide Puppet policy to be implemented on the node. Manifests are the closest thing that could be called a *program* in the conventional sense.

You learned how to create and apply Resources, the smallest building blocks of a Puppet manifest. You learned the common syntax of a resource, and how to retrieve the details of an existing resource on a system in the Puppet configuration language.

You learned how to instruct the Puppet agent to output messages during application of the manifest for debugging purposes.

You also learned how you can execute programs in the policy, but also why this can be bad design. You learned how using a more declarative style uses less code, and works more consistently in all situations.

CHAPTER 5

Using Puppet Configuration Language

Now that you've been introduced to the Puppet manifest and the Resource building block, we are going to introduce you to the Puppet configuration language.

This chapter will introduce you to the data types, operators, conditionals, and iterations which can be used to build configuration policies for Puppet 4 manifests. Writing manifests well is the single most important part of using Puppet. You'll find yourself returning to this chapter again and again as you develop your own manifests.

This chapter is intended to provide a detailed reference for of the configuration language. Don't stop here if you don't understand how or why to use one piece of the language just yet. Just take in what is possible, and refer to this chapter as you build and grow your manifests, and make them into Puppet modules.

First we'll get started with how variables work in Puppet.

Defining Variables

Like many scripting languages, variables are prefaced with a $. The variable name must start with a lower case letter, and may contain lower case letters, numbers, and underscores.

```
$myvar          # valid
$MyVar          # invalid

$my_var         # valid
$my-var         # invalid
$_myvar         # invalid
```

```
$my3numbers     # valid
$3numbers       # invalid
```

Previous versions of Puppet allowed upper case letters, periods, and dashes with inconsistent results. Puppet 4 has improved reliability by enforcing these standards.

Values are assigned with an = sign. The values can be boolean, numbers, strings, arrays, or hashes. As I'm sure you've been introduced to these concepts many times before, we'll jump straight to some examples.

```
$not_true       = false                         # boolean
$num_tokens     = 115                           # number
$my_name        = 'Joe'                         # string
$my_list        = [1,4,7]                       # items to an array
[$first,$last] = ['Jo','Rhett']                 # array to array
$key_pairs      = {name => 'Joe', uid => 1001}  # hash
```

As I'm sure you can guess from the examples, anything after a # mark is a comment ignored by the interpreter.

New to Puppet 4, you can declare the data type of a variable at assignment.

```
Boolean $not_true            = false
Integer $num_tokens          = 115
String $my_name              = 'Joe'
Array[Integer] $my_list      = [1,4,7]
Array[String] [$first,$last] = ['Jo','Rhett']
Hash $key_pairs              = {name => 'Joe', uid => 1001}
```

As this feature has the most benefit for input validation in Puppet modules, we cover this topic extensively in Creating Puppet Modules: Validating Input with Types. For now just be aware that it is possible.

You can find more details about variables and Puppet's native data types at https://docs.puppetlabs.com/puppet/latest/reference/lang_variables.html and https://docs.puppetlabs.com/puppet/latest/reference/lang_datatypes.html.

Defining Numbers

In Puppet 4 unquoted numerals are validated as Numeric data type.

- Decimal numbers start with 1 through 9.
- Floating point numbers contain a single period within them.
- Octal numbers (most commonly used for file modes) start with a `0`.
- Hexadecimal numbers (used for memory locations or colors) start with `0x`.

In previous versions of Puppet, bare numbers were evaluated as unquoted strings. Best practice as of Puppet 3 was to quote all numbers. In Puppet 4 numbers are their own data type which have explicit validation performed against them.

Always quote numbers which may be misinterpreted, such as decimals with leading zeros.

```
$decimal     = 1234          # valid decimal assignment
$octal       = 0775          # valid octal assignment
$hexadecimal = 0xFFAA        # valid hexadecimal assignment
$string      = '001234'      # decimal number with leading zeros
```

Using Variables in Strings

Strings with pure data should be surrounded by single quotes.

```
$my_name = 'Dr. Evil'
$how_much = '100 million'
```

Use double quotes when interpolating variables into strings, as shown in the example below.

```
$the_greeting = "${my_name}, you've been given ${how_much} dollars!\n"
```

Using the correct quotes avoids difficult situations with input data.

```
$num_tokens = '100 million $ dollars'   # US dollars, not a variable to be evaluated
```

Limiting Problems with Brackets

As with most scripting languages, curly brackets should be used to delineate variable boundaries.

```
$the_greeting = "Hello ${myname}, you've been given ${num_tokens} tokens!\n"

notice { 'value2':
  message => "The second value in the list=${my_list[1]}\n",
}
```

Best Practice: Use curly brackets to delineate beginning and end of a variable name within a string.

Use curly brackets any time you use a variable within a string, but not when using the variable by itself. As shown below, the variable is used by itself in the resource, so it reads easier without the brackets.

```
# This time we define the strings in advance
$file_name = "/tmp/testfile2-${my_name}.txt"
$the_greeting = "Hello ${myname}, you've been given ${num_tokens} tokens!\n"

# Don't use brackets for variables that stand alone
file { $file_name:
  ensure   => present,
  mode     => '0644',
  replace  => true,
  contents => $the_greeting,
}
```

No Redefinition

Variables may not be redefined in Puppet within a given namespace, or scope. We'll cover the intricacies of scope in the Part II: Creating Puppet Modules, but understand that a manifest has a single namespace, and a variable cannot receive a new value within that namespace.

This is one of the hardest things for experienced programmers to get used to. However, if you consider the nature of declarative programming, it makes a lot of sense.

In procedural programming, you have a specific order of events and an expected state of change as you pass through the code.

```
myvariable = 10
print myvariable  # prints 10
myvariable = 20
print myvariable  # prints 20
```

However in a declarative language, the interpreter handles variable assignment independently of usage within resources. Which assignment would be performed prior to the resource implementation? In my experience this could change from one node to the other or even one evaluation to another on the same node. To avoid this problem, Puppet kicks out an error if you attempt to change a variable's value.

```
[vagrant@client ~]$ cat double-assign.pp
$myvar = 5
$myvar = 10

[vagrant@client ~]$ puppet apply double-assign.pp
Error: Cannot reassign variable myvar at /home/vagrant/double-assign.pp:2 on node client.example.c
```

Consider this part of the learning process for thinking declaratively.

Finding Facts

Speaking of variables, Facter provides many variables for you containing node-specific information. These are always available for use in your manifests. Go ahead and run the facter program by hand, and look at the output.

```
[vagrant@client ~]$ facter
architecture => x86_64
augeasversion => 1.0.0
blockdevice_sda_model => VBOX HARDDISK
blockdevice_sda_size => 10632560640
blockdevice_sda_vendor => ATA
blockdevices => sda
domain => example.com
facterversion => 2.3.0
filesystems => ext4,iso9660
fqdn => client.example.com
gid => vagrant
hardwareisa => x86_64
hardwaremodel => x86_64
hostname => client
id => vagrant
interfaces => eth0,eth1,lo
ipaddress => 10.0.2.15
...etc
```

You can also use Puppet to list out facts in JSON format:

```
[vagrant@client ~]$ puppet facts find
$ puppet facts find
{
  "name": "client.example.com",
  "values": {
    "puppetversion": "4.1.0",
    "virtual": "virtualbox",
    "is_virtual": true,
    "architecture": "x86_64",
    "augeasversion": "1.3.0",
    "kernel": "Linux",
    "domain": "example.com",
    "hardwaremodel": "x86_64",
    "operatingsystem": "CentOS",
```

As you can see, facter produces a significant number of useful facts about the system. From facts that won't change over the lifetime of a system, like platform and architecture, to information that can change from moment to moment, like memory free. Try the following commands to find some of the more variable fact information provided.

```
[vagrant@client ~]$ facter | grep version

[vagrant@client ~]$ facter | grep mb

[vagrant@client ~]$ facter | grep free
```

Facter can also provide the data in different formats, useful for passing to other programs. The following options output facter data in the common YAML and JSON formats.

```
[vagrant@client ~]$ facter --yaml
[vagrant@client ~]$ puppet facts --render-as yaml

[vagrant@client ~]$ facter --json
[vagrant@client ~]$ puppet facts --render-as json
```

Retrieving Values

Now let's cover how to access and use variables. Each data type has different ways, and sometimes different rules, to access them.

Strings should be surrounded by single quotes. Use double quotes when interpolating variables into strings, as shown in the example below.

```
notice( 'Beginning the program.' )
notice( "Hello ${myname}, glad to see you today!" )
```

Best Practice: Use curly brackets to delineate beginning and end of a variable name within a string.

You can access specific values within an Array or Hash by using the index or key respectively.

```
notice( "The second value in my list is ${my_list[1]}" )
notice( "The name in my key pair hash is $key_pair{'name'}" )
```

As with most scripting languages, curly brackets should be used to delineate variable boundaries. Use curly brackets any time you use a variable within a string, but not when using the variable by itself. Here is an example of using a pre-defined variable properly:

```
# Don't use quotes or brackets around variables that stand alone
file { $file_name:
  ensure   => present,
  mode     => '0644',
  replace  => true,
```

```
  contents => $the_greeting,
}
```

The facts provided by Facter can be referenced like any other variable. The facts are available in a `$facts` array. For example, to customize the message shown on login to each node, use a File resource like this.

```
file { '/etc/motd':
  ensure   => present,
  mode     => '0644',
  replace  => true,
  contents => "Host ${facts['hostname']}, running ${facts['os']['release']['full']}\n",
}
```

When reading older Puppet manifests, you'll find that they refer to facts using an older style with just the fact name such as `$fact name`. This is dangerous as the fact could be overwritten either deliberately or accidentally within the scope the code is operating in. A slight improvement is to refer to the fact explicitly in the top-level scope with `$::factname`. However the variable could be altered or changed there as well. Finally, neither of these inform the code reader whether the value was defined in a manifest, or by a fact.

Best Practice: Refer explicitly to the fact within the `$facts` hash. This ensures you will receive the unaltered output from facter, and informs the reader where the value came from.

Avoiding Reserved Words

As you've seen in our examples, boolean values, strings, and numbers can all be used as bare words, or without quotes.

There are a number of reserved words which have special meaning for the interpreter, and must be quoted when used as string values. These are all fairly obvious words that are reserved within many other programming languages:

- and
- attr
- case
- class
- default
- define
- else
- elsif

- false
- function
- if
- in
- import
- inherits
- node
- or
- private
- true
- type
- undef
- unless

There really aren't any surprises in this list. Any language primitive (as named above), resource type (e.g. file, exec, etc), or function name cannot be used as a bare word string.

You can find a complete list of all reserved words at https://docs.puppetlabs.com/puppet/latest/reference/lang_reserved.html.

There is a legitimate problem where previous working code can break when new functions or resource types are introduced with the same name. It is best to avoid all possibility of failure, by quoting strings every time. We do this in all examples throughout the book.

```
$my_variable = somestring    # valid
$my_variable = 'somestring'  # safer
```

Modifying with Operators

You can use all standard arithmetic operators for variable assignment or evaluation. As before, we're going to provide examples and skip an explanation you've likely learned many times in your life.

```
$added       = 10 + 5       # 15
$subtracted  = 10 - 5       # 5
$multiplied  = 10 * 5       # 50
$divided     = 10 / 5       # 2
$remainder   = 10 % 5       # 0
$two_bits_l  = 2 << 2       # 8
$two_bits_r  = 64 >> 2      # 16
```

If bit shifting operators aren't something you're used to, you can safely ignore them. You don't want to use the bit shift operators unless you're one of us geeks fluent in binary, and know exactly what you are doing.

New in Puppet 4, you can concatenate arrays and merge hashes with +. You might remember these structured data types we defined in the previous section:

```
$my_list = [1,4,7]
$bigger_list = $my_list + [14,17]   # equals [1,4,7,14,17]

$key_pairs = {name => 'Joe', uid => 1001}
$user_definition = $key_pairs + { gid => 500 }  # hash now has name, uid, gid...
```

You can append to arrays with the << operator. Watch out though, as an array appended to an array creates a single entry in the array containing an array in the last position.

```
$my_list << 33            # equals [1,4,7,33]
$my_list << [33,35]       # equals [1,4,7,[33,35]]
```

You can also get the difference between two hashes with the - operator.

```
$hash_one = {name => 'Jo', uid => 1001, gid => 500 }
$hash_two = {name => 'Jo', uid => 1001, home => '/home/jo' }
$difference = $hash_one - $hash_two  # hash with gid and home directory
```

Concatenation, Append, and Difference are new features of Puppet 4 not available in any previous version of Puppet.

We'll cover the comparison operators in the next section about conditionals.

Order of Operations

The operators have the precedence used by standard math and all other programming languages. If you find this statement vague, it is because I deliberately intended it so. Very few people know all the rules for precendence.

Do yourself and whoever has to read your code a favor -- use parenthesis to make the ordering explicit. Explicit ordering is more readable and self-documenting.

```
$myvar = 5 * (10 + $my_var)   # you don't need to know operator precedence to understand this
```

If you are stuck reading someone else's code who didn't use parenthesis, the implicit order of operations is documented at https://docs.puppetlabs.com/puppet/latest/reference/lang_expressions.html#order-of-operations.

Using Conditional Operators

If you have any experience programming, you'll find Puppet's comparison operators familiar and easy to understand. Any expression using comparison operations will evaluate to Boolean `true` or `false`. First, let's discuss all the ways to evaluate statements. Then we'll go over how to use the boolean results.

Number comparisons operate much as you might expect.

```
4 != 4.1                # number comparisons are simple equality match
$how_many_cups < 4      # any number smaller than 4.0 is true
$how_many_cups >= 3     # any number larger than 3.0 is true
```

String operators are a bit inconsistent. String equality comparisons are case insensitive, while substring matches are case sensitive.

```
coffee == 'coffee'      # bare word string is equivalent to quoted single word
'Coffee' == 'coffee'    # string comparisons are case insensitive
$cup_of_joe != 'tea'
'tea' !in 'coffee'      # you can't find tea in coffee
'fee' in 'coffee'       # but you can pay your daily barista fee
'Fee' !in 'coffee'      # substring matches are case sensitive
```

Array and Hash comparisons match only with complete equality of both length and value. The `in` comparison looks for a value matches in arrays, and key matches in hashes.

```
[1,2,5] != [1,2]                # array matching tests for identical arrays
5 in [1,2,5]                    # value found in array

{name => 'Joe'} != {name => 'Jo'}          # hashes aren't identical
'Jo' !in {fname => 'Jo', lname => 'Rhett'} # Jo is a value and doesn't match
```

When doing comparisons you'll find the standard Boolean operators **and**, **or**, and **!** (not) work exactly as you might expect.

```
true and true               # true
true and false              # false
true or false               # true
true and !false             # true
true and !true              # false
```

Except `in`, every one of these operators can be used by those odd people who enjoy Backus Naur form. Yes, you, we know about you. And no, I'm not going to initiate any of these innocent people into your ranks. Either you already know Backus Naur, or you should enjoy your innocence. Just be aware that they work in that form, should you ever find yourself in need of that particular perversion.

You can find a complete list of all operands and operators with example uses at https://docs.puppetlabs.com/puppet/latest/reference/lang_expressions.html.

Creating Regular Expressions

Puppet supports standard Ruby regular expressions, as defined at http://www.ruby-doc.org/core/Regexp.html. The regex operator works with a string on the left, and the regular expression on the right of the ~ operator.

```
$what_did_you_drink =~ /tea/       # likely true if English
$what_did_you_drink !~ /coffee/     # likely false if up late
```

You can use regular expressions in four places:

- Conditional statements: if and unless
- Case statements
- Selectors
- Node definitions (deprecated)

As Regular Expressions are well documented in numerous places we won't spend time covering how to use them here, other than to provide some examples below. You may find O'Reilly's reference book Regular Expressions Pocket Reference very handy.

```
unless $facts['operatingsystem'] !~ /(?i-mx:centos|fedora|redhat)/ {
        include yum
}

case $facts['hostname'] {
  /^web\d/: { include role::webserver  }
  /^mail/ : { include role::mailserver }
  default : { include role::base       }
}

$package_name = $facts['operatingsystem'] ? {
  /(?i-mx:centos|fedora|redhat)/ => 'mcollective',
  /(?i-mx:ubuntu|debian)/        => 'mcollective',
  /(?i-mx:freebsd)/              => 'sysutils/mcollective',
}
```

If you need to truly master regular expressions, there is no better book than O'Reilly Media's Mastering Regular Expressions.

Evaluating Conditional Expressions

Now, let's put these expressions you've learn to use with conditional statements. You have four different ways to utilize the boolean results of a comparison.

- If / Elsif / Else Statements
- Unless / Else Statements
- Case Statements
- Selectors

As you'd expect, there's always the basic if/elsif/else I'm sure you know and love.

```
if ($coffee != 'drunk') {
  notify { 'best-to-avoid': }
}
elsif ('scotch' == 'drunk') {
  notify { 'party-time': }
}
else {
  notify { 'party-time': }
}
```

There is also the `unless` statement which provides the inverse of If. Unless is considered bad form by some, but if it reads easier with `unless` then use the most readable variant. The following example shows why unless can be tricky reading with an else.

```
# The $id fact tells us who is running the Puppet agent
unless( $facts['id'] == 'root' ) {
  notify { 'needsroot':
    message => "This manifest must be executed as root.",
  }
}
else {
  notify { 'isroot':
    message => "Running as root.",
  }
}
```

The use of `else` with `unless` is new to Puppet 4.

The case operator can be used to do numerous evaluations, avoiding a long string of multiple elsif(s). You can test explicit values, match against another variable, use regular expressions, or evaluate the results of a function. The first successful match will execute the code within the block following a colon.

```
case $what_she_drank {
   'wine':            { include state::california }
   $stumptown:        { include state::portland   }
   /(scotch|whisky)/: { include state::scotland   }
   is_tea( $drink ):  { include state::england    }
   default:           {}
}
```

Always include a bare word `default` option when using `case` statements, even if the default does nothing as with my example above.

`Selectors` are similar to case statements however they return a value instead of executing a block of code. This can be useful when defining variables. A selector looks like a normal assignment however the value to be compared is followed by a question mark and a block of comparisons with fat commas identifying the matching values.

```
$native_of = $what_he_drinks ? {
   'wine'            => 'california',
   $stumptown        => 'portland',
   /(scotch|whisky)/ => 'scotland',
   is_tea( $drink )  => 'england',
   default           => 'unknown',
}
```

As a value must be returned in an assignment operation, a match is required. Always include a bare word `default` option with a value.

So the drinking comparisons have been fun, but let's examine some practical comparisons that you may actually use in a real manifest.

```
# Explicit comparison
if( $facts['osfamily'] == 'redhat' ) {
   include yum
}
# Do a substring match
elsif( $facts['osfamily'] in 'debian-ubuntu' ) {
   include apt
}
# New package manager is only available with FreeBSD 9 and above
elsif( $facts['operatingsystem'] =~ /?i:freebsd/ ) and ( $facts['os']['release']['major'] >= 9 ) {
   include pkgng
}
```

This can be more compact as a case statement:

```
case $facts['osfamily'] {
  'redhat':                                          { include yum   }
  'debian', 'ubuntu':                                { include apt   }
  'freebsd' and ($facts['os']['release']['major'] >= 9) { include pkgng }
  default: {}
}
```

There's also `unless` to reverse comparisons for readability purposes.

```
unless $facts['kernel'] == Linux {
   notify { 'You are on an older machine.': }
}
else {
   notify { 'We got you covered.': }
}
```

Selectors are also useful for handling heterogenous environments.

```
$libdir = $facts['osfamily'] ? {
  /(?i-mx:centos|fedora|redhat)/ => '/usr/libexec/mcollective',
  /(?i-mx:ubuntu|debian)/        => '/usr/share/mcollective/plugins',
  /(?i-mx:freebsd)/              => '/usr/local/share',
}
```

You can find a complete list of all conditional statements with more example uses at https://docs.puppetlabs.com/puppet/3.7/reference/lang_conditional.html.

Building Lambda Blocks

Lambdas are blocks of code which accept parameters to be passed in. You can think of them as functions without a name. You will use Lambas with the iterator functions such as `each` introduced in the next section to perform a set of instructions on multiple values. If you are experienced with Ruby Lambdas, you'll find the syntax to be identical.

Lambdas are a new, advanced feature of Puppet 4 not available in any previous version of Puppet.

A lambda begins with one or more variable names between pipe `|` `|` operators. These name the variables which will be passed into the block of code.

```
| $firstvalue, $secondvalue | {
  block of code which operates on these values.
}
```

The lambda has its own variable scope. This means that the variables named between the pipes exist only within the block of code. You can name these variables any name you want, as they will be filled by the values passed by the function into the lambda on each iteration. Other variables within the context of the lambda are also available, such as local variables or node facts.

For example, here is a policy which will output the list of disk partitions from the hash provided by Facter. Within the loop we refer to the *hostname* fact on each itera-

tion. The device name and a hash of values about each device are stored in the `$name` and `$device` variables during each loop.

```
$ puppet apply /vagrant/manifests/mountpoints.pp
each( $facts['partitions'] ) |$name, $device| {
  notice( "${facts['hostname']} has device ${name} with size ${device['size']}" )
}

$ puppet apply /vagrant/manifests/mountpoints.pp
Notice: Scope(Class[main]): Host geode has device sda1 with size 524288
Notice: Scope(Class[main]): Host geode has device sda2 with size 3906502656
Notice: Scope(Class[main]): Host geode has device sdb1 with size 524288
Notice: Scope(Class[main]): Host geode has device sdb2 with size 3906502656
```

As shown above, you must enclose array and hash index references within brackets to ensure that you output only the value requested. Otherwise it will output the entire array or hash, followed by the literal text in brackets.

Next we'll cover each of the functions which can iterate over values and pass them to a lambda.

Looping through Iterations

In this section we're going to introduce powerful new functions for iterating over sets of data. You can use iteration to evaluate many items within an array or hash of data using a single block of code (a lambda, described on the previous page).

Iterations are a new feature of Puppet 4 not available in any previous version of Puppet.

To give you can idea of how powerful and important iteration is, let's discuss what we used to do in previous versions of Puppet.

I wrote a number of modules that did smart things based on what networks a node was connected to for my consulting clients. In Puppet 3, it was necessary to write code that would make guesses about which interfaces were available on the system, and then check each one. Or try to discover the interface names in reverse using the IP data to tell us which interface they were bound too. It worked within a limited sense, but remained prone to failure in any environment which didn't tightly match

the network and hard configuration of my consulting client. This is not how Puppet modules should be written.

With Puppet 4 the modules have been rewritten to iterate through all available interfaces and IP addresses, removing all guesswork or statically configured interface names from the modules.

Here are some practical examples available to you from the basic facts provided by Facter. However you can use iteration with any data point which can be presented as an array or a hash.

- Going through all IP addresses assigned to a node to see if any meet a specific condition.
- Going through all users on a node to find ones which match a certain criteria.
- Going through all mounted partitions to determine the total space available to the node.

There are five functions that iterate over a set of values and pass each one to a lambda for processing. The lambda will process each input and return a single response containing the processed values. Here are the five functions, what they do to provide input to the lambda, and what they expect the lambda to return as a response.

- `each` invokes the lamba once for each entry in an array, or each key-value pair in a hash.
- `filter` provides a filtered subset of the array or hash containing only entries which were matched by the lambda.
- `map` returns a new Array or Hash from the results of the lambda
- `reduce` compiles an array or hash to a single value which is passed to the lambda
- `slice` creates small chunks of an array or hash and passed it to the lamba

The following examples show how these functions can be invoked. They can be invoked like traditional functions:

```
each( $facts['partitions'] ) |$name, $device| {
  notice( "${facts['hostname']} has device $name with size ${device['size']}" )
}
```

You can also chain function calls to the values they operate on, which is a common usage within Ruby:

```
$facts['partitions'].each() |$name, $device| {
  notice( "${facts['hostname']} has device $name with size ${device['size']}" )
}
```

Finally, new to Puppet 4 is the ability to use a hash or array literal instead of a variable. The following example demonstrates iteration over a literal array of names:

```
['sally','joe','nancy','kevin'].each() | $name | {
  notice( "$name wants to learn more about Puppet." )
}
```

Now let's review each of the functions which can utilize a lambda.

Each

The each function invokes a lambda once for each entry in a array, or each key-value pair in a hash. The lambda can do anything with the input value, as no response is expected. each is most commonly used to process a list of items.

```
# Output a list of interfaces which have IPs
split( $facts['interfaces'] ).each |$interface| {
  if( $facts["ipaddress_${interface}"] != '' ) {
    notice( "Interface ${interface} has IPv4 address $facts["ipaddress_${interface}"]" )
  }
  if( $facts["ipaddress6_${interface}"] != '' ) {
    notice( "Interface ${interface} has IPv6 address ${facts["ipaddress6_${interface}"]" )
  }
}
```

If you want a counter for the values, providing an array with two entries gives you an index on the first one. For example, creating a list of all the interfaces on a system which hosts virtualization clients yields:

```
$ cat /vagrant/manifests/interfaces.pp
split( $facts['interfaces'], ',' ).each |$index, $interface| {
  notice( "Interface #${index} is ${interface}" )
}
$ puppet apply /vagrant/manifests/interfaces.pp
Notice: Scope(Class[main]): Interface #0 is br0
Notice: Scope(Class[main]): Interface #1 is br0_3
Notice: Scope(Class[main]): Interface #2 is br0_4
Notice: Scope(Class[main]): Interface #3 is br0_7
Notice: Scope(Class[main]): Interface #4 is eth0
Notice: Scope(Class[main]): Interface #5 is eth1
Notice: Scope(Class[main]): Interface #6 is lo
Notice: Scope(Class[main]): Interface #7 is virbr0
Notice: Scope(Class[main]): Interface #8 is virbr0_nic
Notice: Scope(Class[main]): Interface #9 is vnet0
Notice: Scope(Class[main]): Interface #10 is vnet1
Notice: Scope(Class[main]): Interface #11 is vnet2
Notice: Scope(Class[main]): Interface #12 is vnet3
Notice: Compiled catalog for geode.netconsonance.com in environment production in 0.54 seconds
Notice: Finished catalog run in 0.07 seconds
```

Be aware that the index is placed in the first variable, and the array entry in the second. This is the opposite of the same concept used in Ruby and ERB templates provided by `each_with_index`, where the array entry comes in the first variable and the index in the second.

You can also use `each` on hashes. If you provide a single variable you'll get an array with two entries. If you provide two variables you'll have the key in the first one, and the value in the second one.

```
$ cat /vagrant/manifests/uptime.pp
each( $facts['system_uptime'] ) |$type, $value| {
  notice( "System has been up ${value} ${type}" )
}

$ puppet apply /vagrant/manifests/uptime.pp
Notice: Scope(Class[main]): System has been up 23:04 hours uptime
Notice: Scope(Class[main]): System has been up 83044 seconds
Notice: Scope(Class[main]): System has been up 23 hours
Notice: Scope(Class[main]): System has been up 0 days
Notice: Compiled catalog for client.example.com in environment production in 0.26 seconds
Notice: Finished catalog run in 0.07 seconds
```

The following manifest would provide the exact same results.

```
each( $facts['system_uptime'] ) |$uptime| {
   notice( "System has been up $uptime[1] $uptime[0]" )
}
```

Each returns the result of the last operation performed. In most cases, you'll use `each` to process each entry and you won't care about the return, however this could be useful if the value of the last entry has some meaning for you. While you might consider calculating an aggregate value from the operations, that is exactly what the `reduce` function is for.

Filter

The `filter` function returns a filtered subset of an array or hash containing only entries which were matched by the lambda. The lambda block evaluates each entry and returns a positive result if the item matches.

For an extended example, let's examine all interfaces and find all RFC1918 IPv4 and RFC4291 IPv6 internal-only addresses. We do this with multiple steps:

1. Filter all facts to find the facts containing IP addresses.
2. Filter the first result list to find which contain addresses which internal-only or link-local addresses.

3. Iterate over the second results with `each` and extract the interface name to display with the address.

```
$ips = $facts.filter |$key,$value| {
  $key =~ /^ipaddress6?_/
}
$private_ips = $ips.filter |$interface, $address| {
  $address =~ /^(10|172\.(1[6-9]|2[0-9]|3[0-1])|192\.168)\./
}
$private_ips.each |$ip_interface,$address| {
  $interface = regsubst( $ip_interface, '^ipaddress6?_(\w+)', '\1' )
  notice( "interface $interface has private IP $address" )
}
```

If you apply this on a node you'll get results like this:

```
$ puppet apply /vagrant/manifests/ipaddresses.pp
Notice: Scope(Class[main]): interface virbr0 has private IP 192.168.122.1
Notice: Scope(Class[main]): interface br0_7 has private IP 172.27.1.31
```

At the time this book was written, Facter did not return IPv6 link-local addresses in the results. Hopefully FACT-605 will be fixed by the time you read this, and you'll see IPv6 link-local addresses in the output.

Map

`map` returns a new Array or Hash from the results of the lambda. You call `map` on an array or hash, and it returns a new array containing the results. The lambda should return only values which will be within the new array.
Here's an example where we create an array of IPv4 addresses. We pass in an array of interface names. We use the interface name to look for an IP address associated with that interface name in the `ipaddress_` facts.

As with `filter`, when you pass in an array, the named variable contains the array value.

```
$ips = split( $facts['interfaces'] ).map |$interface| {
  $facts["ipaddress_${interface}"]
}
```

You can also pass in a hash. In this case the named variable will contain an array with the key in the first position and the value in the second. The following example uses `filter` to create a hash of interfaces which have IP addresses. Then it uses `map` to create separate arrays of the interfaces, and the IPs.

```
$ints_with_ips = $facts.filter |$key,$value| {
  $key =~ /^ipaddress_/
}
```

```
# Create an array of ints with IPv4 addresses
$ints = $ints_with_ips.map |$intip| {
  $intip[0] # key
}

# Create an array of IPv4 addresses
$ips  = $ints_with_ips.map |$intip| {
  $intip[1] # value
}
```

Reduce

The `reduce` function processes an array or hash and returns only a single value. It takes two arguments: an array or hash and an initial seed value. If the initial seed value is not supplied, it will use the first entry in the array or hash as the initial seed value. The lambda should be written to perform aggregation, addition, or some other function which will operate on many values and return a single value.

The way reduce utilizes the first entry in the array could have unintended consequences if the entry is not the appropriate data type for the output. As this is a common confusion, we'll show you an example of this problem first.

In the following example we pass the hash of partitions in to add together all of their sizes. As with all other functions, each hash entry is passed in as a small array of `[key,value]`.

```
$ cat /vagrant/manifests/partitions.pp
$total_disk_space = $facts['partitions'].reduce |$total, $partition| {
   notice( "partition $partition[0] is size $partition[1]['size']" )
   $total + $partition[1]['size']
}
notice( "Total disk space = ${total_disk_space}" )

$ puppet apply /vagrant/manifests/partitions.pp
Notice: Scope(Class[main]): partition sdb2 is size 3906502656
Notice: Scope(Class[main]): partition sda1 is size 524288
Notice: Scope(Class[main]): partition sda2 is size 3906502656
Notice: Scope(Class[main]): Total disk space = 7814053888
Total disk space = [sdb1, {filesystem => linux_raid_member, size => 524288}, 3906502656, 524288, 3
```

As we didn't supply an initial value, the first entry is a small array containing the first key, value pair from the hash (sdb1). We then performed addition to add integers to this array, which produced the confusion you see above.

To resolve this situation you should seed the initial value with the appropriate data type (integer 0 in this case). The first hash entry is then processed by the block, adding the integer size to the seed value, creating the output we were looking for.

```
$ cat /vagrant/manifests/partitions.pp
$total_disk_space = $facts['partitions'].reduce(0) |$total, $partition| {
   notice( "partition $partition[0] is size $partition[1]['size']" )
   $total + $partition[1]['size']
}
notice( "Total disk space = ${total_disk_space}" )

$ puppet apply /vagrant/manifests/partitions.pp
Notice: Scope(Class[main]): partition sdb1 is size  524288
Notice: Scope(Class[main]): partition sdb2 is size 3906502656
Notice: Scope(Class[main]): partition sda1 is size 524288
Notice: Scope(Class[main]): partition sda2 is size 3906502656
Notice: Scope(Class[main]): Total disk space = 7814053888
```

Slice

The `slice` function creates small chunks of a specified size from an array or hash. This is perhaps one of the subtlest, and trickiest of the functions to use, as the output changes depending on how you invoke it.

If you invoke `slice` with a single parameter specified between the pipe operators, the value passed into the lambda will be an array containing the number of items specified by the slice size. The following example should make this clear:

```
[1,2,3,4,5,6].slice(2) |$number| {
   notice( "First number is $number[0]" )
   notice( "Second number is $number[1]" )
}
```

If you invoke `slice` with the same number of parameters as the slice size, each variable will contain one entry from the slice. The following example will make this clear:

```
[1,2,3,4,5,6].slice(2) |$one, $two| {
   notice( "First number is $one" )
   notice( "Second number is $two" )
}
```

Unlike the other functions, hash entries are **always** passed in as a small array of `[key,value]`, no matter how many parameters you use. So if you have a slice of size two from a hash, the lambda will receive two arrays, each containing two values: the key and the value from the hash entry. Here's an example that demonstrates the idea.

```
$facts['partitions'].slice(2) |$part1, $part2| {
   notice( "partition names in this slice are $part1[0] and $part2[0]" )
}
```

Similar to `each`, most invocations of slice do not return a value and thus the result can be ignored.

With

The `with` function invokes a lambda exactly one time, passing the variables provided as parameters. The lambda can do anything with the input values, as no response is expected.

You might quite rightly point out that this function doesn't iterate and thus doesn't belong in this section of the book. You're quite right, but I've included it here as it behaves exactly like these other iterators, and can be quite useful for testing.

```
with( 'austin', 'powers', 'secret agent' ) |$first,$last,$title| {
  notice( "A person named ${first} ${last}, ${title} is here to see you." )
}
```

The `with` function is most commonly used to isolate variables to a private scope, unavailable in the main scope's namespace.

Captures-Rest Parameters

Most of the functions only produce one or two parameters for input to a lamba, however `slice` and `with` can both send an arbitrary number of parameters to a lamba. For ease of definition, you can proceed the final parameter with a splat or `*` to indicate that it will accept an array of all remaining parameters.

```
$hostsfile_lines.each.split(' ') |$ipaddr, $hostname, *$aliases| {
  $alias_display = join($aliases,',')
  notice( "Host ${hostname} has IP ${ipaddr} and aliases ${alias_display}" )
}
```

Summary

As you have seen in this section, the functions which iterate over arrays and hashes provide a tremendous amount of power not available in any previous version of Puppet.

You can invoke these functions like traditional functions or by chaining the functions to the data they are processing.

You can find more information about iteration and lambdas at https://docs.puppetlabs.com/puppet/latest/reference/experiments_lambdas.html.

Conclusion

This chapter introduced you to the following components of the Puppet Configuration Language:

- data types can be numbers, strings, boolean (true/false), arrays (or indexes), and hashes (key/value pairs).
- operators perform addition, subtraction, multiplication, division, concatenation and merges of values.
- conditional operators compare data for equality and inclusion.
- regular expressions matches ranges of values or substrings within a value.
- conditional expressions such as if/else and unless/else allow you to limit policy application.
- case and select statements evaluate a value to determine the appropriate action or assignment.
- lambdas are unnamed functions intended to process values passed in by iteration fuctions.
- iterations loop over arrays or hashes performing an operation on each entry, optionally passing to a lambda.

These are the data types and functions available for evaluating and operating on data for use in Resource definitions within a Puppet manifest.

CHAPTER 6

Controlling Resource Processing

You can control how Puppet utilizes and acts upon resources with *metaparameters*. Metaparameters are common attributes which can be used with any resource, including both built-in and custom resource types. Metaparameters control how Puppet deals with the resource.

You can find all metaparameters documented at https://docs.puppetlabs.com/references/latest/metaparameter.html.

Adding Aliases

The `alias` attribute allows you to provide a friendly name to a resource. This is identical to using a friendly name in the title. The alias parameter can be used when the resource title is declared in a less friendly way. Either one allows you refer to the aliased name instead of the resource title.

For example, the following two resources are identical:

```
# alias declared in resource title
mailalias { 'postmaster-alias':
  ensure    => 'present',
  name      => 'postmaster',
  recipient => ['root'],
  target    => '/etc/aliases',
}

# name defaults to resource title, alias provides friendly name
mailalias { 'postmaster':
  ensure    => 'present',
  recipient => ['root'],
  target    => '/etc/aliases',
  alias     => 'postmaster-alias',
}
```

Preventing Action

The noop flag allows you to say that changes to this resource should not be made. For example, if you'd like to know if a newer version of the Puppet package is available without actually performing the upgrade, you could use this syntax.

```
package { 'puppet':
  ensure  => latest,
  noop    => true,
}
```

When this manifest is processed, if a new package version is available it will update the statistic for unprocessed resources. You'll see a message like the following in the logs:

```
Notice: /Stage[main]/Main/Package[puppet]/ensure: current_value 4.00, should be latest ['4.01'] (n
```

Auditing Changes

The `audit` flag will log a message any time an audited resource is changed. This could be useful if you don't want to manage the content of a file, but do want to know every time the content changes.

```
file { '/etc/hosts':
  audit => true,
  noop  => true,
}
```

Defining Loglevel

The `loglevel` attribute allows you to identify what loglevel changes to this resource should be logged at. For example, if you'd like to know if a newer version of the Puppet package is available without actually performing the upgrade, you could use this syntax.

The log levels are identical to syslog loglevels, and map to those when logging on Unix and Linux systems.

1. debug
2. info (also called verbose)
3. notice
4. warning
5. err
6. alert
7. emerg

8. crit

Warn us whenever Puppet is upgraded.

```
package { 'puppet':
  ensure   => latest,
  loglevel  => warning,
}
```

Limiting by Tags

Tags can be used for selective enforcement of resources. This allows you to apply only part of a policy, such as adding packages, without applying other parts of the policy, such as restarting or stopping services. Let's look at any example of this.

Tags can be added to resources as a single string, or as an array of strings. The following policy will tag both the Package and the Service with the *puppet* tag, but also put a *packages* tag on the Package resource.

```
package { 'puppet':
  ensure => present,
  tag    => ['package','puppet'],
}

service { 'puppet':
  ensure => running,
  enable => true,
  tag    => 'puppet',
}
```

If you run this manifest it will start the Puppet agent, which perhaps isn't desirable right now. So you can apply the policy and limit action to resources marked with the *package* tag.

```
[vagrant@client ~]$ sudo puppet apply /vagrant/manifests/packagetag.pp --tags package
Notice: Compiled catalog for client.example.com in environment production in 0.71 seconds
Notice: Finished catalog run in 0.26 seconds

[vagrant@client ~]$ puppet resource service puppet
service { 'puppet':
  ensure => 'stopped',
  enable => 'false',
}
```

As you can see, the policy was applied however the puppet service was not started. This demonstrates the power of the tags to limit policy evaluation on demand.

It wasn't necessary to add a tag of 'package' to the package resource. Puppet automatically adds a tag of the resource type to each and every resource. We could have run the policy with `--tags service` and affected only the service, even though the 'service' tag isn't explicitly list in the definition above.

The `--tags` command line option can accept multiple comma-separated tags. So you could invoke the same recipe with `--tags package,service` to process both of them without processing resources of a different type.

Limiting by Schedule

A schedule can be used to limit when Puppet will make changes to a resource. For example, if we want to limit upgrades of Puppet until after the normal working day has ended, we might declare it this way.

```
package { 'puppet':
  ensure   => present,
  schedule  => 'after-working-hours',
}
```

The value of the `schedule` metaparameter must be the name of a Schedule resource you've declared. Let's start with some examples.

```
schedule { 'business-hours':
  period  => hourly,
  repeat  => 1,                              # apply once an hour
  range   => '08:00 - 17:00',                # between 8am and 5pm
  weekday => ['Mon','Tue','Wed','Thu','Fri'], # on weekdays
}

schedule { 'after-working-hours':
  period      => daily,
  repeat      => 2,               # apply no more than twice a day
  range       => '17:00 - 08:00',  # between 5pm and 8am
}
```

The period can be any of the following values.

1. hourly
2. daily
3. weekly
4. monthly
5. never

The `repeat` attribute limits how many times it will be applied with the period. The default is 1.

You can find the complete documentation for the Schedule resource at https://docs.puppetlabs.com/references/latest/type.html#schedule.

Defining Resource Defaults

You can declare defaults for all resources of a given type. If a resource declaration of the same type does not explicitly use the attribute, then the default value for that attribute will be used.

Defaults are declared with a capitalized resource name and no title. For example, the following resource definition would make all Packages be applied after working hours.

```
Package {
  schedule => 'after-working-hours',
}
```

Conclusion

This chapter introduced you to attributes that control how resources are processed. Those attributes are as follows:

- *aliases* provide easy to use names for resources with complicated titles.
- *noop* can be used to prevent changes to the resource.
- *audit* can be used to log all changes to a resource without affecting it.
- *loglevel* can be used to control log output on a per-resource basis.
- *tags* can be used to perform limited runs which affect only certain resources in a manifest.
- A *schedule* can be used to limit when changes to a resource are permitted.
- Default values for attributes can be defined for any resource which doesn't override them.

Using these attributes provides fine-grained control over how and when your resources are updated.

CHAPTER 7

Expressing Relationships

This chapter focuses on metaparameters which can create and manage relationships between resources.

The relationship between resources control which resources will be processed prior to other resources. Puppet uses this information to build a dependency graph of relationships between resources, and thus process each in the appropriate order.

Resource relationships and ordering are perhaps the most confusing things to people learning Puppet. Most people are familiar with linear processing, controlled by the order expressed within the file. Puppet provides metaparameters to define dependencies to be handled within and between manifests. This is significantly more powerful than rigid, linear ordering for the following reasons:

1. Linear ordering is easy to write once, but hard to extend.
2. Linear ordering prevents the ability for one piece of code to easily extend another piece of code.
3. Linear ordering provides a strict chain of processing, whereas expressed relationships allow for multiple dependencies.
4. Many to one relationships are harder to learn, but considerably more powerful.
5. Loose ordering prevents accidental dependencies, allowing more resources to be processed if a single dependency is not met.

You will come to understand the power and flexibility of Puppet's resource ordering when you build a module that extends a community-provided module. For now, simply keep in mind that Puppet will process the resources by evaluating the dependency graph created from the metaparameters introduced in this chapter.

Managing Dependencies

There are situations where avoiding implicit dependencies of linear ordering can provide significant value.

In a linear ordering (e.g. not Puppet-like) dependency evaluation, every succeeding statement is assumed to depend on the statement before it. In that case, nothing else in that script should be processed as they might depend on the statement which failed.

In a scenario where the manifest has 6 operations listed in order **A** -> **B** -> **C** -> **D** -> **E** -> **F**, if A fails then should all of B through F not happen?

This would be undesirable in Puppet, if a single resource early in the manifest was not essential to a significant portion of the manifest. By allowing you to explicitly declare dependencies in resources, Puppet can enforce significantly more of the catalog. In the example shown above, it may be that only step F depends on A, so resources B through E can be processed.

During the puppet catalog compilation, Puppet will evaluate the dependencies for each resource. If a dependency for a resource fails, neither it nor any resource which depends on it will be processed during the current Puppet run. This is generally desirable behavior.

Puppet's explicit dependency metaparameters provide for complex and powerful dependency management. Let's show you how to use them.

Referring to Resources

As shown throughout all previous examples, you declare a resource using the resource type lowercase and enclose the definition in curly brackets.

```
package { 'puppet':
  ensure => present,
}
```

Once the resource has been given a unique title, it is possible to refer to that resource by name. This is called a *Resource Reference*. In this chapter we're going to refer to specific resources quite often, so let's describe how to do it. To create a resource reference, uppercase the first letter of the resource type and enclose the title in square brackets. For example, when referring to the package resource created above, you'd use `Package['puppet']`. Here an example which builds a service to run the puppet agent installed above.

```
service { 'puppet':
  ensure  => running,
  enabled => true,
```

```
  require => Package['puppet'],
}
```

So remember: create a resource with the lowercase type, and refer to an existing resource with an uppercase first letter.

An easy way to remember this is the common name versus proper name rule of English. A park is a resource type, but Golden Gate Park is a specific instance... e.g. a proper noun, the first letter of which is always capitalized.

Ordering Resources

In many situations some resources must be applied before others. For example, you cannot start a service until after you install the package which contains the application. Here we will show you the `before` and `require` metaparameters you can use to ensure the package is installed before the service is started.

```
package { 'puppet':
  ensure  => present,
  before  => Service['puppet'],
}

service { 'puppet':
  ensure  => running,
  enable  => true,
  require => Package['puppet'],
}
```

The `before` and `require` metaparameters are redundant in this case. Either one would work by itself. Use the one which fits your manifest and is easiest to read. Belt and suspenders people like myself often use both when possible.

Ordering resources can be a trap. Many Puppet novices try to order every resource into a strict pattern, no matter whether the resources are truly dependent or not. This makes an implementation very fragile and likely to fail when extended. Think *less is more* list only the necessary dependencies.

Triggering Refresh Events

The `before` and `require` metaparameters ensure that dependencies are processed before resources that require them. However, these parameters do not link or provide data to the other resource.

The `notify` and `subscribe` metaparameters operate in a similar manner, but will also send a refresh event to the dependent resource if the dependency is changed. The dependent resource will take a resource-specific action. For example, a service would restart after the configuration file has been changed.

Let's modify our previous policy to upgrade the Puppet package whenever a newer version is available.

```
package { 'puppet':
  ensure  => latest,
  notify  => Service['puppet'],
}

service { 'puppet':
  ensure  => running,
  enable  => true,
  subscribe => Package['puppet'],
}
```

In this case, if a newer version of Puppet is available then the puppet package will be upgraded. Any time the Puppet package is installed or upgraded, the puppet service will be restarted.

As before, the `notify` and `subscribe` metaparameters are redundant. Either one would send the refresh event without the other. Belt and suspenders people like myself do both.

The refresh event means something special to Exec resources. If an Exec resource is created with the attribute `refreshonly` set to true, then the Exec resource will not run unless it receives a refresh event. In the following example, we will update the facts.yaml file for MCollective only after Puppet has been upgraded.

```
Package { 'puppet':
  ensure => latest,
  notify => Exec['update-facts'],
}

exec { 'update-facts':
  path        => ['/bin','/usr/bin'],
  command     => 'facter --yaml > /etc/mcollective/facts.yaml',
  refreshonly => true,
}
```

Under normal conditions this Exec resource will not execute. However, if the Puppet package is installed or upgraded, the Notify attribute will send a refresh event to the Exec and the command will be run.

Chaining Resources with Arrows

You can also order and related resources using Chaining Arrows. You put the required resource on the left, and the dependent resource on the right, linked together with `->`. For example, to install puppet before starting the service you could express it like so.

```
Package['puppet'] -> Service['puppet']
```

You can use `~>` to also send a refresh event, like `notify` does. For example, this will restart the Puppet service after the Puppet package is upgraded.

```
Package['puppet'] ~> Service['puppet']
```

The chaining arrow syntax is harder to read than the metaparameters, and should be avoided when possible. In particular, right to left relationships are harder to read and explicitly against the Puppet Style Guide.

```
# Don't do this. Order it left -> right instead.
Service['puppet'] <~ Package['puppet']
```

Processing with Collectors

A collector is a grouping of many resources together. You can use Collectors to affect many resources at once.

A Collector is declared by the capitalized type followed by `<|`, an optional attribute comparison, and `|>`. Let's examine some collectors.

```
User <||>                       # every User
User <| groups == 'wheel' |>    # Users in the wheel group declared in a manifest
Package <||>                    # every Package
Package <| tag == 'packages' |> # Packages tagged with 'packages' tag in a manifest
Service <||>                    # every Service
Service <| enabled == true |>   # Services set to start at boot time in a manifest

# Search expressions may be grouped with parenthesis and combined
Service <| ( ensure == running ) or ( enabled == true ) |>     # Services running OR set to start
Service <| ( ensure == running ) and ( title != 'puppet' ) |>  # Services other than Puppet set to
```

Note the key words *declared in a manifest*. The user resource mentioned in the first example would only match users who are declared in a manifest to be within the 'wheel' group, and not a user which was added to the group outside of Puppet. To refer back to what we know about how Manifests are processed, the resources that are matched by a Collector must be explicitly declared within the catalog compiled from the manifest(s).

One place where chaining arrows have proven very useful is when processing many resources with Collectors. By combining chaining arrows with Collectors, you can set dependencies for every resource of one type.

For example, you could have our previous exec update the Facts whenever any package is added or removed.

```
# Regenerate the facts whenever a package is added, upgraded, or removed
Package <||> ~> Exec['update-facts']
```

Likewise, you could ensure that the Puppet Labs yum repository is installed before any packages tagged with *puppet* or *mcollective*.

```
Yumrepo['puppetlabs'] -> Package <| tag == 'puppet' |>
Yumrepo['puppetlabs'] -> Package <| tag == 'mcollective' |>
```

Best Practice: Limit use of collectors to clearly scoped and limited effect. A collector which matches all resources of a given type will affect a resource a team member adds to the policy next week or month, not realizing there is a collector in a manifest he or she did not modify. The best usage of collectors affects only the resources within the same manifest.

You can find more details about Collectors at https://docs.puppetlabs.com/puppet/latest/reference/lang_collectors.html.

Understanding Puppet Ordering

During the catalog compilation, prior to implementation of any resources, Puppet creates a dependency graph containing all resources with dependencies declared using the metaparameters and chaining arrows discussed in this chapter. It uses this graph to determine what order to implement the resources.

Resources without explicit ordering parameters are not guaranteed to be ordered in any specific way. In versions of Puppet greater than 2.6, unrelated resources were evaluated in an order which was apparently random, but was consistent from run to run. (In versions of puppet prior to 2.6 it was not consistent from node to node or run to run.) The only way to ensure that one resource was implemented before another was to define dependencies explicitly.

An agent configuration option `ordering` was introduced in Puppet 3.3 which allowed control of ordering for unrelated resources. This configuration option accepts three values:

title-hash (default in all previous versions of Puppet)
: Orders unrelated resources randomly but consistently between runs.

manifest (default in Puppet 4)
: Orders unrelated resources by the order they are declared in the manifest.

random
: Orders resources randomly and changes the order on each run. This is useful for identifying missing dependencies in a manifest.

Although resources in a manifest will generally be implemented in the order defined, never count upon implicit dependencies. Always define all dependencies explicitly. This is especially important when utilizing or extending another manifest or module, or when your manifest or module could be extended by someone else. *Hint: always*

Best Practice: State all dependencies explicitly.

You can flush out missing dependencies by testing your manifests with the `random` ordering option. Each time you run the following, the resources will be ordered differently. This almost always causes failures for any resources missing necessary dependencies.

```
$ puppet apply --ordering=random testmanifest.pp
```

Conclusion

Puppet 4 will implement the resources in a manifest according to a dependency graph created from the following explicit dependency controls:

- `before` metaparameter and the `->` chaining arrow
- `notify` metaparameter and the `~>` chaining arrow
- `require` metaparameter
- `subscribes` metaparameter

Puppet 4 will implement resources not listed in the dependency graph in the order they are declared. This ordering can be changed using the agent configuration option `ordering`. Never depend on the manifest ordering, instead declare all relationships explicitly.

The `random` ordering option is useful for testing manifests for missing dependencies.

CHAPTER 8

Upgrading Puppet 3 Manifests

The upgraded parser used in Puppet 4 makes several changes to the Puppet language. This section concerns itself with changes from Puppet 3. If you are new to Puppet, you can safely skip this section until and unless you find yourself upgrading older Puppet code.

Validating Numbers

In previous versions of Puppet numbers were really just strings. Unquoted numbers were unquoted strings, which happened to work most of the time. Documented best practice in Puppet 3 was to quote all numbers to ensure they were explicitly strings, but a lot of people (the author included) have modules with unquoted numbers in them.

In Puppet 4 unquoted numerals are a Numeric data type. Numbers are validated as part of the catalog compilation, and an invalid number will cause the catalog to fail to compile.

- Decimal numbers start with 1 through 9.
- Floating point numbers contain a single period within them.
- Octal numbers (most commonly used for file modes) start with a 0.
- Hexadecimal numbers (commonly used for memory locations or colors) start with 0x.

To avoid catalog compilations it is best to quote numbers which may be misinterpreted, such as decimals with leading zeros.

```
$decimal     = 1234          # valid decimal assignment
$octal       = 0775          # valid octal assignment
$hexadecimal = 0xFFAA        # valid hexadecimal assignment
$string      = '001234'      # decimal assignment with leading zeros
```

Any unquoted string which starts with a number is likely to be validated as if it were a number, and may cause a compile failure. All of the following will cause errors during the parsing phase.

```
$leading_zeros = 0991    # will be mistaken for octal and raise error
$mixed_chars = 0x1x2x    # will be mistaken for hexadecimal and raise error
$color_code = 0xFFAh     # the suffix will cause validation failure and raise error
```

As mentioned elsewhere, the easiest way to avoid confusion is to always quote strings. As you can pass a string containing a number as input to anything which will accept a number, it is useful to quote numbers which may be misinterpretted.

File Modes are not Numbers

Although it would be fantastic to use the new number validation with file modes, an unfortunate decision[1] was made to require all file modes to be strings.

This decision means that modules which had unquoted numbers for the mode will throw errors, rather than implement an unexpected file rights set. I understand the reasoning, but I would have greatly preferred to use the implicit number validation for file modes in Puppet 4.

Using Hash and Array Literals

Older versions of Puppet required you to assign arrays and hashes to variables before using them in resources or functions. Puppet 4 allows you to use literal arrays and hashes more naturally in a Puppet manifest.

```
notify { "(['one','two','three'][1] )": }    # produces the output "two"
```

New in Puppet 4, you can concatenate arrays and merge hashes with +

```
$my_list = [1,4,7]
$bigger_list = $my_list + [14,17]                # equals [1,4,7,14,17]

$key_pairs       = {name => 'Joe', uid => 1001}
$user_definition = $key_pairs + { gid => 500 }   # hash now has name, uid, gid...
```

Also new in Puppet 4, you can append to arrays with <<. Watch out though, as an array appended to an array creates a single entry in the array containing an array in the last position.

```
$my_list << 33            # equals [1,4,7,33]
$my_list << [33,35]       # equals [1,4,7,[33,35]]
```

1 https://tickets.puppetlabs.com/browse/PUP-2156 comments contain decision to force all file modes to string

Adding Else to Unless

You can now use `else` with `unless`.

```
unless ( $somevalue > 10 ) {
   # do this
} else {
   # do that
}
```

It is generally better to use `if/else` than `unless` for readability, but readable code is the most important thing. If it reads easier with `unless` then use it.

Chaining Assignments

You can now change multiple assignments in the same expression. You can chain both equality and addition operations, like so:

```
$security_deposit = 100
$first = 250
$last = 250
$down_payment = $first + $security_deposit

# could be rewritten as
$first = $last = 250
$down_payment = $first + ( $security_deposit = 100 )
```

Chained assignments have low precedence, so it's necessary to use parenthesis to ensure proper ordering.

Best Practice: As you can see from the example, chained assignments are rarely more readable than the expanded version. Avoid for documentation sake.

Expressions Can Stand Alone

Previous versions of Puppet required the results of all expressions to be assigned. In Puppet 4 an expression can stand alone. If it is the last expression in a block of code (e.g. a function), the result will be returned. Otherwise the value will be discarded.

The following is perfectly valid as the last expression within a block of code.

```
$myvar * 10
```

Blocks of code / functions return the result of their last expression. You do not need an explicit `return`.

You can safely call a function without using the returned value. Earlier versions of Puppet would raise errors in this situation.

Chaining Expressions with a Semicolon

You can now use a semicolon to concatenate two expressions together.

```
$fname = 'Jo'; $lname = 'Rhett'
```

The semicolon can be used to prevent the last statement of a block from being returned as the value.

```
{
  $fname = 'Jo'; $lname = 'Rhett'; 1     # returns 1 for success
}
```

Best Practice: Using a semicolon to change operations never makes more readable code. Leave this for special case situations where you must execute several commands on a single line.

Calling Functions in Strings

You can now call functions from within a double-quoted string.

```
notify { 'need_coffee':
  message => "I need a cup of coffee. Come remind me in ${fqdn_rand(10)} minutes.",
}
```

Improved Error Reporting

Error reports are much improved with Puppet 4. You'll generally see the following improvements:

1. Some errors now show position on line, using `filename:line_number:character_number`.

2. Many block and token parsing errors which gave confusing direction about what to fix have been improved.

3. You can now change the maximum amount of errors and warnings Puppet will output before giving up. The following settings in *puppet.conf* will override the defaults of 10 for each.

```
[main]
  max_errors = 3
  max_warnings = 3
  max_deprecations = 20
```

No matter what the limits are, a final error lists the error and warning count.

Avoiding Upgrade Problems

The improved parser in Puppet 4 has cleaned up many consistency issues from previously unclear documentation. If you are writing manifests, you should adopt these practices immediately to avoid upgrade problems. If you maintain older manifests, you should go through them and address these issues. **Every change mentioned in this section is backwards compatible and will work with Puppet 3 and earlier.**

Variable names are now limited to lowercase letters, numbers, and underscores. Variable names must contain at least one letter.

```
$5 = 'hello'         # invalid
$5letters = 'hello'  # valid
```

Parameters in classes and defines have the same limitations:

```
define mytype( $21, $32 ) {             # invalid
define mytype( $twenty1, $thirty2 ) {   # valid
```

Bare words must start with a lowercase letter. You'll see fairly common usage of unquoted values `present`, `true`, etc. Unquoted values (bare words) must start with a lowercase letter, not a number.

```
$myvar = fourguys     # valid
$myvar = 4guys        # invalid
$myvar = some4guys    # valid
$myvar = '4guys'      # quoted strings are always safest
```

Unquoted numbers will be validated. An error is raised if an unquoted number is not valid. The following are situations where unquoted numbers would work in Puppet 3 but cause errors in Puppet 4.

```
# hexadecimal
$address =  0x1AH     # error 'H' is not a valid hex number
$address = '0x1AH'    # safe as a string
$address = 0x1A       # safe number decimal 26

# leading zeros cause numbers to evaluated as octal
$leadingzero = 0119         # error octal has 8 bits, 0-7
$leadingzero = '0119'       # safe as quoted string

# octal numbers are commonly used for file modes
$mode =  1777         # sticky bit directory missing leading 0 will evaluate decimal
$mode = 01777         # valid sticky bit world writable (e.g. /tmp)
```

Don't forget the exception that in Puppet 4 the file and template resources won't accept numeric modes. You must use a string, like so:

```
file '/etc/testfile.txt' {
   mode => '0644',
}
```

Deprecations

There are some features which were popular or necessary in Puppet 2 days which have been deprecated in Puppet 3 and are completely gone from Puppet 4. All of these have been known about for years, so there should be no surprises here:

- The ability to write Puppet manifests in pure Ruby has been removed. It never worked very well in the first place.
- The limited and broken `puppet kick` is gone, replaced by a much more powerful and extendable MCollective agent for Puppet.
- The `import` statement is gone. This was never a good idea. Best practice is to put the code in a module class, and include the class.
- Node inheritance was the old way to apply classes to groups of similar nodes. This has been completely obsoleted by Hiera assignment of classes.

CHAPTER 9

Conclusion of Part I

In this section you've created a safe, risk-free learning environment you can utilize to write and test Puppet Manifests. You've learned the following things about Puppet policies:

- Puppet policies are written in Manifests.
- Manifests contain one or more Resource declarations.
- Resources create, alter, or remove their types: users, groups, files, etc.
- Facter provides data about the node useful for local customization.

In this Part you have learned each part of the Puppet Configuration Language and how to utilize it to create manifests. You've used Puppet Apply to implement the manifest on your test system. Puppet Apply:

- Parses a Puppet manifest file and reports any errors.
- Utilizes Facts about the system as variables for customization.
- Executes immediately on the local system.
- Provides verbose output informing you of what it has done.

While many people utilize Puppet Apply only for testing manifest changes, it can be used at broad scale if a method of synchronizing the manifests to each node is available. We'll ways to implement this, and the pros and cons of this approach in Part IV: Puppet Server.

Best Practices for Writing Manifests

Before you move on to the next chapter, I'd like to remind you of best practices for writing Puppet Manifests:

- Quote the resource title.
- `ensure` should be the first attribute in a resource block.
- Align the arrows for attributes within a resource block.
- Group resources by their relationship with each other.
- Don't use conditionals within resource declarations.
- Provide defaults for case and select statements.
- When it can be done multiple ways, always use the most readable.

You can find the Puppet Style Guide at https://docs.puppetlabs.com/guides/style_guide.html. All of the examples in this book have been compliant with the style guide.

Continued Learning

To expand on what you have learned in this chapter, investigate the built in resources provided by Puppet. There are more than we have discussed in this chapter, and you'll find many of them immediately useful. Here are just a few we didn't mention previously.

augeas	a programatic API for managing configuration files
cron	crontab entries
host	host file entries
interface	networking
mailalias	mail aliases
mount	filesystem mount points
nagios_*	type to manage nagios host, service, contact entries
router	manages a connected router
sshkey	SSH key management
yumrepo	package repository

The complete list of built-in resources can be found at https://docs.puppetlabs.com/references/latest/type.html

PART II

Creating Puppet Modules

Puppet modules are bundles of Puppet code, files, templates, and data. Well-written Puppet modules provide a clean interface for sharing re-usable code between different teams either within your organization, or throughout the global community.

Puppet modules provide the following benefits:

- Organize code and data within the module's namespace.
- Execute one or more manifests.
- Provide files, templates, tests, functions, and plugins.

In this part we'll discuss how to find and use Puppet modules that other people have made available for you. We'll show you how to provide data to these modules such that you can use them without modifying the module code.

While you may find old examples of Puppet manifests used independently, it has been Best Practice for many years now for all manifests to reside within modules. We will go through the process of turning the manifests built in Part I into fully formed and well-built Puppet Modules.

CHAPTER 10

Creating a Test Environment

Puppet provides the ability to serve clients different versions of modules and data using *Environments*. You can use environments to provide unique catalogs to different groups of machines, which can be very useful in large environments with many teams.

However a primary usage of environments is to provide the ability to test out changes to Puppet policy without breaking production environments.

In my opinion, Puppet environments are so necessary and useful for code testing and deployment that I am not going to discuss using Puppet without environments. So before we go on to install Puppet modules, let's set up *production* and *test* environments.

Verifying the Production Environment

The default environment used by Puppet clients is named *production*. For this reason every Puppet installation should have a Puppet environment named production, even if it is not used. You should find that this already exists.

```
[vagrant@client ~]$ ls -l /etc/puppetlabs/code/environments/production
total 4
-rw-r--r-- 1 root root 879 Mar 26 19:27 environment.conf
drwxr-xr-x 2 root root   6 Mar 26 19:38 manifests
drwxr-xr-x 2 root root   6 Mar 26 19:38 modules
```

This environment already has an environment configuration file, and directories for modules and manifests.

Creating a Test Environment copy

Now let's create a *test* environment you can use to test out new modules, or changes to modules, prior to implementing them in production. Create the environment like so:

```
[vagrant@client ~]$ mkdir -p /etc/puppetlabs/code/environments/test/modules
```

Now we have a place to test module changes prior without breaking any production nodes. Now, let's go one step farther and enable a reminder that we are using the testing environment.

```
[vagrant@client ~]$ mkdir /etc/puppetlabs/code/environments/test/manifests
[vagrant@client ~]$ $EDITOR /etc/puppetlabs/code/environments/test/manifests/site.pp
```

Make the contents of this file something like this:

```
notice( "Processing catalog from the Test environment." )
```

This will give you a warning any time you use this environment. This warning can be a helpful reminder to move a node back to the production environment when testing is complete.

Changing the Base Module Path

There is a convenience setting which allows you to share modules between all environments.

```
[main]
  environmentpath = /etc/puppetlabs/code/environments
  basemodulepath = /etc/puppetlabs/code/modules
```

The `environmentpath` variable contains a path under which a directory for each environment will be created.

The `basemodulepath` variable contains a directory which will be used as a fallback location for modules not found in the environment's modules directory. This allows you to place common and well-tested modules in a single location shared by all environments.

Both of the directory names shown are the default values, and do not need to be specified in the configuration file.

Skipping Ahead

While there are many things you can fine tune and customize with Puppet environments, many of them wouldn't make much sense at this point in our learning process. The environments are ready for us to use.

We'll come back to cover environments in much greater detail in Customizing Environments just a few more chapters later.

CHAPTER 11

Separating Data from Code

An important part of using modules is to separate the code from the input data. A module written for a single target node may work fine with explicit data within the code; however it won't be usable on other systems without changes to the code.

If the data resides within the code, you'll find yourself constantly going back to hack `if/then/else` conditions into the code for each necessary difference. I'm sure you've done this before, or may even have to do this now to maintain scripts you use today. This chapter will introduce a better way.

Moving the data (values) out of the code (manifest) creates reusable blocks of code which can implement policy in a flexible manner.

Introducing Hiera

Hiera is a key/value lookup tool for configuration data. Hiera is integrated seamlessly into Puppet to provide dynamic lookup of configuration data for Puppet manifests.

Hiera allows you to provide node-specific information to a Puppet module. Hiera uses a customizable hierarchy to lookup the data. This allows you to customize how information is structured within your organization.

For example, at a small company you may organize your data in this way:

1. Company-wide defaults
2. Operating system specific changes
3. Site-specific information

A much large organization might have a hierarchy like the following:

1. Operating system specific configurations

2. Enterprise-level defaults
3. Company specifics
4. Division overrides
5. Production / Staging / QA / Development
6. Region (US, EU, Asia)-specific changes
7. Cluster-specific changes
8. Alterations based on applications deployed to the node

Complex multi-level hierarchies make it easy to utilize the same shared code throughout a diverse organization.

Creating Hiera Backends

Hiera has two built-in data file backends: **YAML** and **JSON**, and then the **Puppet** data provider.

Each of these backends support four data types:

1. Strings
2. true/false (Boolean)
3. Arrays
4. Hashes

Let's go through how to utilize these data types in each backend.

Hiera Data in YAML

The easiest and most common way to provide data to Hiera is utilizing the YAML file format. Files must be have a `.yaml` file extension.

Files in YAML format always start with three dashes by themselves on the first line. The YAML format utilizes indentation to indicate the relationships between data. YAML should always be written using spaces, never tabs, for indentation.

Here are some examples of strings, boolean, arrays, and hashes in YAML.

```
# string
agent_running: 'running'

# boolean
agent_atboot: true

# array
puppet_components:
  - facter
  - puppet

# a hash of values
```

```
puppet:
  ensure: 'present'
  version: '4.0.1'

# A variable loopup
hostname: %{facts::hostname}
```

As this data is all about managing the Puppet agent, why don't we organize this within a single hash? That could look as simple as this:

```
puppet:
  agent_running: 'running'
  agent_atboot: true
  components:
    - 'facter'
    - 'puppet'
```

As you can see, YAML provides a clean, easy to read way to provide data without too much syntax. You can find out more about YAML at http://www.yaml.org/YAML_for_ruby.html.

It is not always necessary to quote strings in YAML. The words 'running', 'facter', and 'puppet' above would have been correctly interpreted as strings without the quotes. However the rules for when you must quote strings in YAML are many and often subtle, so I quote them to avoid possible misinterpretations.

Hiera Data in JSON

You can also provide data to Hiera with the JSON file format. Files must have a `.json` file extension.

As is common with almost every use of JSON, the root of each data source must be a single hash. Each key within the hash names a piece of configuration data. Each value within the hash can be any valid JSON data type.

Our previous example rewritten in JSON format would look like the following. This example shows values of a string, a boolean, and an array of strings:

```
{
  "puppet": {
    "agent_running": "running",
    "agent_atboot": true,
    "components": [
      "facter",
      "puppet"
    ]
  }
}
```

You can find complete details of the JSON data format at http://www.json.org/.

Hiera Data in Puppet

A Puppet backend for Hiera is simply the name of a Puppet class in which Hiera will check for data variables. These variables would be defined the same as in any Puppet manifest.

Here is a simple example matching our previous two.

```
class 'data' {
  # Configuration hash for Puppet
  $puppet = {
    "agent_running" => "running",
    "agent_atboot"  => true,
    "components"    => ['facter', 'puppet'],
  }
}
```

Hiera receives every Puppet variable and can use it without performing a lookup, so the Puppet data source is completely unnecessary for a simple example as shown above. The Puppet data source should only be enabled when the module is performing complex operations on the lookup values to produce answers dynamically.

Puppet Variable and Function Lookup

You can lookup Puppet variables or execute functions to interpolate data within a Hiera value. Interpolation is performed on any value prefixed with a `%` and surrounded by brackets `{}`.

To return the value of a puppet variable, just place the variable name within the brackets. For example: `%{facts::hostname}`

Functions can be invoked within the interpolation brackets as well: `%{ split([1,2,3]) }`.

Configuring Hiera

Puppet looks for a Hiera configuration file at the location specified by the `hiera_config` configuration variable. By default this is *$codedir/hiera.yaml*, or */etc/puppetlabs/code/hiera.yaml* in Puppet.

If you utilize Hiera command line tools or perform Hiera lookups in pure Ruby code, they expect to find the configuration file in */etc/hiera.yaml*. It is common to symbolically link this to the configuration file in the puppet configuration directory.

The Hiera configuration file is in YAML format. Each top-level item is a Ruby Symbol prefaced with a colon. Valid Symbol names are alphanumeric with underscore, but not containing dashes. There will always be at least three top-level items. Let's go over these now.

Backends

The configuration key `:backends` should provide an array which lists the backend data providers that Hiera should use. There are three built-in backends, the two data types we discussed previously, **YAML** and **JSON**, plus Hiera can utilize data from Puppet.

If you wish to utilize both built-in file types, you could configure it as follows.

```
:backends:
  - yaml
  - json
```

We will only be utilizing YAML within this book.

Other Hiera plugins can be used to provide new data backends to Hiera, however that is beyond the scope of this book.

Backend Configuration

For each backend data provider you name in the `backends` array, you should create a top-level entry with the name of the provider. For each backend provide a hash of configuration data.

For the two built-in file-based backends the only configuration key necessary is `:datadir`, which identifies the directory in which the data files reside.

```
:yaml:
  :datadir: /etc/puppetlabs/code/environments/%{::environment}/hieradata
:json:
  :datadir: /etc/puppetlabs/code/environments/%{::environment}/hieradata
```

As the files read by each backend must be named differently, you can use the same data directory for both data sources as shown above.

You'll note that we're using the top-level environment variable (defined by puppet master or client) to allow different environment data in each environment. Let's go ahead and create the hieradata directory now in the environment directories we created in the last chapter.

```
[vagrant@client ~]$ mkdir /etc/puppetlabs/code/environments/test/hieradata
[vagrant@client ~]$ mkdir /etc/puppetlabs/code/environments/production/hieradata
```

For the **Puppet** backend place the name of a Puppet module which contains the data in a `:datasource` configuration key.

```
:puppet:
  :datasource: hieradata
```

With this configuration variables from a module named `heiradata` would be accessed for Hiera lookups. As mentioned previously this is only useful when the module provides dynamic lookup of data.

Logger

By default Hiera logs warning and debug messages to STDOUT. You can change this using the `:logger` configuration value. Valid values are

- Console: emit warnings and debug on STDERR (default)
- noop: don't emit messages
- The name of a Ruby class named Hiera::*name*_logger which provides **warn** and **debug** class methods.

Note that this value is only used for command line tools and direct Ruby access. Puppet overrides the value and logs Hiera messages utilizing the Puppet internal logger.

Hierarchy

The final mandatory parameter is **:hierarchy**. The hierarchy defines the priority order for lookup of configuration data. For single values Hiera will proceed through the hierarchy until it finds a value and then stop. For arrays and hashes Hiera will merge data from each level of the hierarchy, selecting the winner of conflicts based on the `:merge_behavior` configuration setting.

There are two types of data sources: static and dynamic. Static data sources are files explicitly named in the hierarchy which contain data. Dynamic data sources are files which are named using interpolation of local configuration data, such as the hostname or operating system of the node.

In a larger enterprise the data lookup hierarchy could be quite complex, however I recommend the following for a good starting point.

1. Put default values in a file named *global.yaml*.
2. Put all operating system specific information in a file named for the OS family as returned by Facter, e.g. *RedHat.yaml*, *Debian.yaml*, *FreeBSD.yaml*, etc.

3. Put information specific to a single node within a file named the full hostname of the node with a *.yaml* extension.

You would implement this hierarchy using the following configuration syntax. As you can see, we are interpolating data provided by Facter to choose which files will be read.

```
:hierarchy:
  - defaults
  - "%{::hostname}"
  - "%{::osfamily}"
  - global
```

Naturally you can extend this hierarchy to use information like the domain name of the node or any other facter-provided node value.

If you have multiple backends configured, then Hiera will evaluate the entire hierarchy for the first configured backend, then evaluate the entire hierarchy in order for the 2nd configured backend, etc.

Merge Behavior

You can use the global `:merge_behavior` configuration parameter to change the way that data at different levels of the hierarchy is merged.

You should avoid changing this until you have created a testbed for evaluating the behavior before and after the change.

The value must be one of the following:

native (default)	Merge keys only. Values at the higher priority will return the entire array or hash intact from that priority level.
deeper	Merge array and hash values recursively. If a recursive key conflicts, the higher priority value overrides the lower priority value.
deep	Merge array and hash values recursively. If a recursive key conflicts, the lower priority value overrides the higher priority value.

The *deep* choice is counter-intuitive and reverse of expectations. You don't want this, unless you really want to bring the pain.

Anything but the default **native** requires the `deep_merge` Ruby gem to be installed.

Complete Example

Following is a complete example of a Hiera configuration file. This example is what we will use for the remainder of this book. It enables YAML data input from */etc/puppetlabs/code/hieradata*, with a hierarchy that uses host-specific information in preference to operating system family information, finally defaulting to values global to every host.

```
---
:backends:
  - yaml
:yaml:
  :datadir: /etc/puppetlabs/code/hieradata
:hierarchy:
  - defaults
  - "%{facts::clientcert}"
  - "%{facts::osfamily}"
  - global
```

Let's go ahead and install this file now in your Puppet configuration directory.

```
[vagrant@client ~]$ sudo mkdir /etc/puppetlabs/code/hieradata
[vagrant@client ~]$ sudo cp /vagrant/etc-puppet/hiera.yaml /etc/puppetlabs/code/hiera.yaml
```

Doing Hiera Lookups in a Manifest

Let's modify one of our manifests to utilize Hiera data. First, let's create a manifest which contains variables for the configuration of the Puppet agent service.

```
# Always set a default value when performing a Hiera lookup
$running = hiera( 'puppet::running', 'running' )
$startatboot = hiera( 'puppet::atboot', true )

# Now the same code can be used regardless of the value
service { 'puppet':
  ensure => $running,
  enable => $startatboot,
}
```

Now, let's create a Hiera data file containing values for this service.

```
$ sudo $EDITOR /etc/puppetlabs/code/hieradata/global.yaml
```

Within this file place the following values:

```
---
puppet:
  running: 'running'
  atboot: true
```

Finally, let's execute our manifest utilizing the Hiera data.

```
$ sudo puppet apply hierasample.pp
```

Testing Hiera Lookups

There are two ways to test hiera lookups from the command line: using the `--config` command line argument to specify the hiera configuration file, like so:

```
[vagrant@client ~]$ hiera --config /etc/puppetlabs/code/hiera.yaml key...
```

Or you can provide hiera with its own configuration file, or better yet symbolically link one to the other.

```
[vagrant@client ~]$ sudo ln -s /etc/puppetlabs/code/hiera.yaml /etc/hiera.yaml
```

With either of these options you can do Hiera lookups from the command line to test your data sources:

```
[vagrant@client ~]$ hiera puppet::running
running
[vagrant@client ~]$ hiera puppet::packages
["hiera", "puppet"]
```

CHAPTER 12

Using Modules

One of the most powerful benefits of using Puppet is having access to the shared community of module developers. While Puppet provides the ability to create modules to do nearly anything, it is quite possible that somebody has already written the module you need. You may be able to use it intact, or use it as a starting point for further development.

In this chapter we'll go over how to find, evaluate, install, and use modules provided by Puppet Labs and the global community.

Finding Modules

Puppet Forge

Internal Forge https://forge.example.com/

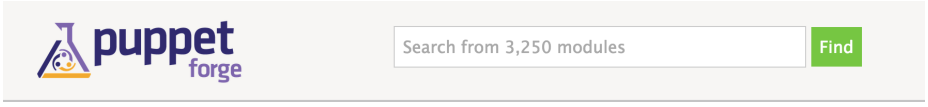
Puppet Forge

The single largest repository of Puppet modules is the Puppet Forge.

The Puppet Forge provides an easy to use interface to search for and find Puppet modules provided by others. It is also the default repository used by the `puppet module` command. Let's go over how to use it.

Start by opening a browser and going to https://forge.puppetlabs.com/. The very first thing you'll see is the search interface:

Figure 12-1. Puppet Forge Search



Enter the name of an application you wish to manage with Puppet into the search interface, and you'll receive a list of modules which may do what you are looking for. For example, enter **apache** into the search query to see what modules exist to manage the Apache HTTPd server.

You can also search the Puppet Forge from the command line. For example, the following search will provide the same results.

```
$ puppet module search apache
Notice: Searching https://forgeapi.puppetlabs.com ...
NAME                DESCRIPTION                              AUTHOR        KEYWORDS
puppetlabs-apache   Installs, configures, and manages  ...   @puppetlabs   apache web ssl rhel
example42-apache    Puppet module for apache                 @example42    apache example42
evenup-apache       Manages apache including ajp proxy ...   @evenup       apache vhost ajp
theforeman-apache   Apache HTTP server configuration         @theforeman   apache foreman httpd
snip many other results
```

I have personally found the puppet module command-line search useful when I am trying to remember the name of a module that I have already researched. However the web interface provides a lot of useful information not available in the command line output.

Public GitHub Repositories

Many people share their Puppet modules on GitHub. The following search will show you a significant number of modules which may not be found in the Puppet Forge: https://github.com/search?q=puppet.

A great many of the modules available on the Puppet Forge are also available on GitHub. If you click on the **Project URL** or **Report Issues** links in the Puppet Forge, the vast majority of the time you will find yourself on the GitHub repository for the module in question. If you need to report a problem, or provide a suggested improvement to the module, you'll find yourself using GitHub extensively.

There are a number of reasons that authors may not have published modules available on GitHub to the Puppet Forge.

- They haven't done the work to prepare and bundle the module appropriately for the Puppet Forge. (The author of this book falls into this category.)
- They do not feel the module is appropriate or ready for general purpose use.

- The module does not meet the licensing requirements for publication on the Puppet Forge. Be careful to read the license file provided by any module on GitHub!

Internal Repositories

If you work within a larger organization, you may also have an internal repository of modules. Depending on the software used to provide the forge, the web search interface will vary. However you can use the stock `puppet module` command with any forge:

```
$ puppet module search --module_repository=http://forge.example.org/ apache
```

If you always or exclusively use the internal forge, you can add this parameter to your puppet.conf file to simplify command line searches.

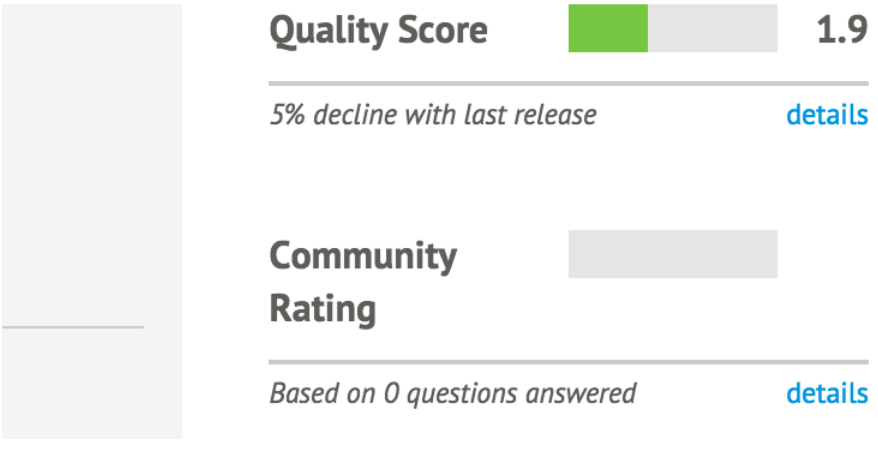
```
[main]
  module_repository = http://forge.example.org/
```

Evaluating Module Quality

While there are many high-quality modules on the Puppet Forge or GitHub, not all modules are created equal. It is best to examine the modules carefully before implementing them in your environment.

The Puppet Forge indicates some information about each entry on the right side of the page. It shows the results of both automated tests of the code base, and community feedback on the module.

Figure 12-2. You might want to look carefully at a module with a score like this.

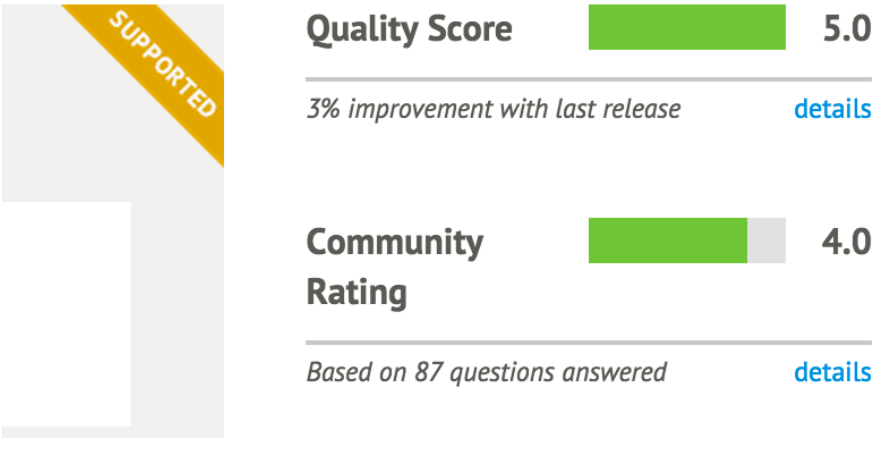


Let's review some ways to identify high-quality modules.

Puppet Supported

Puppet **Supported** Modules are modules which are written by, and offically supported by Puppet Labs.

Figure 12-3. This module is fully supported by Puppet Labs.



From the Puppet Labs website:

> Puppet Labs guarantees that each supported module:

- has been tested with Puppet Enterprise
- is subject to official Puppet Labs Puppet Enterprise support
- will be maintained over the lifecycle, with bug or security patches as necessary
- is tested on and ensured compatible with multiple platforms

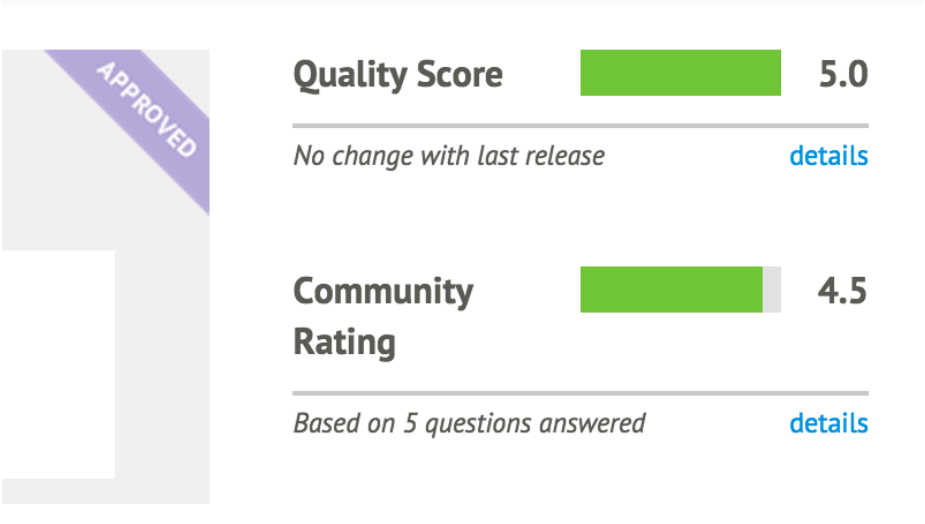
—https://forge.puppetlabs.com/supported

From personal experience, these modules work very well for base use cases. However there are not very many Support modules, and they tend not to handle highly customized situations.

Puppet Approved

Puppet **Approved** modules are modules which have been reviewed by Puppet Labs, and meet their standards for quality.

Figure 12-4. This module has been reviewed and approved by Puppet Labs.



From the Puppet Labs website:

> Puppet Labs ensures that Puppet Approved modules:
>
> - Solve a discrete automation challenge
> - Are developed in accordance with module best practices
> - Adhere to Puppet Labs' standards for style and design
> - Have accurate and thorough documentation to help you get started quickly
> - Are regularly maintained and versioned according to SemVer rules

- Provide metadata including license, issues url, and where to find source code
- Do not deliberately inject malicious code or otherwise harm the system they're used with

—https://forge.puppetlabs.com/approved

In my personal experience, I have not found a module which Puppet Labs has approved which is of low quality, or where I was unable to get minor problems in the module fixed in a reasonable amount of time.

Quality Score

The *Quality Score* of a module is the result of automated review and testing of the module.

Figure 12-5. You might want to look carefully at a module with a score like this.

You can see details of the test results by clicking on *Details* next to the score then scrolling down the page, or by scrolling down and clicking *Scores* just below the module version.

Figure 12-6. A breakdown of the quality test results.

At the time this book was written, a click on the *Details* link produced no visible effect. It did change the page to display the *Scores* section farther down the page, but this was not visible on many screens notably laptop displays. You have to click the link and then scroll down to see the changes.

As you can see the Quality Score is broken down into three tests:

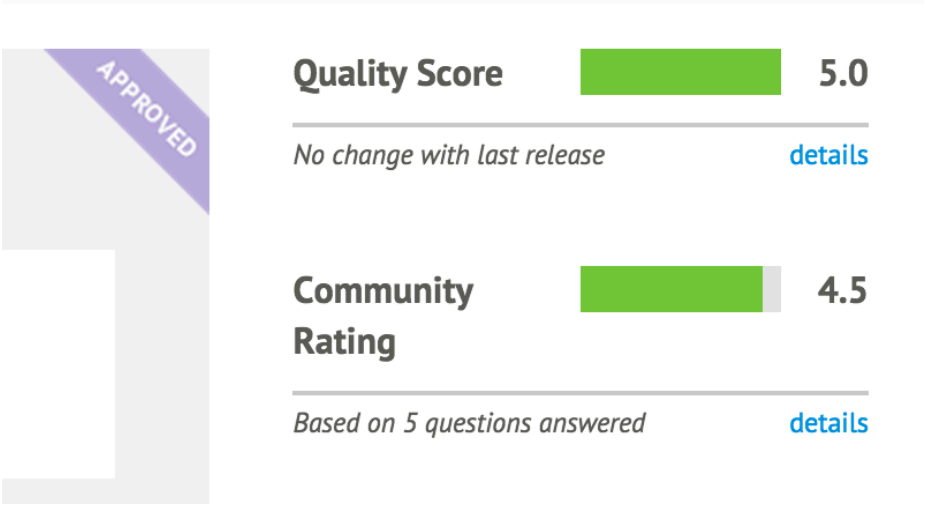
1. Code Quality
2. Puppet Compatibility
3. Metadata Quality

You are provided with a review of the issues found by each test, and a link to see detailed test results.

Community Rating

The final piece of data provided to you about a module is the *Community Rating*. This provides you information about what other users thought of the module.

Figure 12-7. This module received 4.5 out of 5 (e.g. 9 out of 10) score.



You can see details of the community rating by clicking on *Details* next to the rating then scrolling down the page, or by scrolling down and clicking *Scores* just below the module version. You'll find the Community Rating just below the Quality Score.

Figure 12-8. Breakdown of Feedback Rating

As with the Quality Score, at the time this book was written a click on the *Details* link produced no visible effect. You have to click the link and then scroll down to see the changes.

Installing Modules

You can install Modules from the Puppet Forge, a private internal Forge, or directly from the developer's code repository.

Installing from a Puppet Forge

The process for installing a module from the Puppet Forge (or an internal forge of your choice) is very simple. Let's go ahead and do this to install a very useful module that many other modules depend upon: the Puppet Labs-support StdLib module.

```
[vagrant@client ~]$ puppet module install puppetlabs-stdlib
Notice: Preparing to install into /home/vagrant/.puppet/code/modules ...
Notice: Created target directory /home/vagrant/.puppet/code/modules
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/home/vagrant/.puppet/code/modules
└── puppetlabs-stdlib (v4.5.1)
```

As you'll note, this command installed the module into the Vagrant user's private *.puppet* directory. This is an excellent place for you to examine the module and determine if it meets your needs. You can build tests and run them as yourself, to avoid the consequences of a testing failure run by root on an innocent system.

Once you decide that a module will do what you need, you can install this module in the test environment you created at the beginning of this chapter.

```
$ sudo mv .puppet/modules/stdlib /etc/puppetlabs/code/environments/test/modules/
```

If you are certain of the module you are downloading, you can use a different command to install the module directly into the test environment.

```
[vagrant@client ~]$ puppet module install --modulepath=/etc/puppetlabs/code/environments/test/modu
$ puppet module install --modulepath=/etc/puppetlabs/code/environments/test/modules puppetlabs-std
Notice: Preparing to install into /etc/puppetlabs/code/environments/test/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/code/environments/test/modules
└── puppetlabs-stdlib (v4.5.1)
```

At the time this book was written there was a bug preventing this command from working if you were not root. Until PUP-2012 is fixed, you may need to use `sudo` when supplying an explicit modulepath.

Installing from GitHub

Sometimes you will find a module you'd like to use on GitHub, or perhaps you need to test bleeding-edge changes to the module without waiting for the author to update the Puppet Forge. If that's the case, then going straight to their source tree may be your best bet.

If you haven't installed git already, you should do that now.

```
[vagrant@client ~]$ sudo yum install -y git
...snip...
Installed:
  git.x86_64 0:1.8.3.1-4.el7

Dependency Installed:
  libgnome-keyring.x86_64 0:3.8.0-3.el7       perl-Error.noarch 1:0.17020-2.el7       perl-Git.n

Complete!
```

Now we can pull directly from any Puppet module available in a Git repository. For example, if you'd like to get the latest changes to my MCollective module, you can install it from GitHub like so:

```
[vagrant@client ~]$ cd /etc/puppetlabs/code/environments/test/modules
[vagrant@client modules]$ git clone https://github.com/jorhett/puppet-mcollective mcollective
Initialized empty Git repository in /etc/puppetlabs/code/environments/test/modules/mcollective/.gi
remote: Counting objects: 183, done.
Receiving objects: 100% (183/183), 51.13 KiB, done.
remote: Total 183 (delta 0), reused 0 (delta 0), pack-reused 183
Resolving deltas: 100% (98/98), done.
```

Do read the instructions that come with the module. Some modules published on GitHub require steps to be taken before the module can be used successfully, and this module is no exception. Better yet, set it aside for now as there will be an entire chapter on how to use this module in part IV of this book.

Testing a Single Module

In order to test a module you will need to follow the instructions on the page where you found the module. In most situations you will need to

1. Add the module to node's module run list.

2. Define Hiera data keys under the module's name

Let's go through this process now. Start by installing and configuring the *puppetlabs-ntp* module. As this module will change system files, we'll need to run this as root so we must install the module in the system-wide module path.

```
[vagrant@client ~]$ cd /etc/puppetlabs/code/environments/test/modules
[vagrant@client modules]$ puppet module install --modulepath=. puppetlabs-ntp
Notice: Preparing to install into /etc/puppetlabs/code/environments/test/modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/code/environments/test/modules
└─┬ puppetlabs-ntp (v3.3.0)
  └── puppetlabs-stdlib (v4.5.1)
```

If you look in the */etc/puppetlabs/code/environments/test/modules* directory now you'll find both `ntp` and `stdlib` modules installed. The `puppet module install` command will gather all dependency modules listed in the module's metadata.

Looking at the documentation at https://forge.puppetlabs.com/puppetlabs/ntp, you'll find that this module can operate without any input data. It will define the configuration using module defaults. Let's try that out now.

```
[vagrant@client ~]$ puppet apply --environment test --execute 'include ntp'
Notice: Compiled catalog for client.example.com in environment test in 0.79 seconds
Notice: /Stage[main]/Ntp::Config/File[/etc/ntp.conf]/content: content changed '{md5}7fda24f62b1c7a
Notice: /Stage[main]/Ntp::Service/Service[ntp]/ensure: ensure changed 'stopped' to 'running'
Notice: Finished catalog run in 0.54 seconds
```

As you can see the module has modified our ntp configuration, and started the NTP service running. The module works!

Defining Config with Hiera

Now, let's say that you want to change the NTP configuration that the module generated. For instance, the NTP service was configured to only allow connections from localhost. Let's say we want to expand that to allow connections from other systems on the same LAN.

At this point you need to define some test data in Hiera. Let's go ahead and open up the *global.yaml* file in our environment's hieradata directory.

```
[vagrant@client ~]$ $EDITOR /etc/puppetlabs/code/environments/test/hieradata/global.yaml
```

Looking at the documentation at https://forge.puppetlabs.com/puppetlabs/ntp, you'll find that this module's security can be changed by the `restrict` and `interface` parameters.

To provide data for a module's input, the data must be named `modulename::paramname`. So start by defining the `restrict` and `interface` options in the YAML file.

```
---
ntp::interfaces:
ntp::restrict:
```

The documentation says that both of these parameters expect array values. Based on what we learing in Configuring Hiera we use leading dashes to indicate an array. We use single quotes to surround unparsed strings. So the input data for the ntp module would look like this:

```
---
# Data for the puppetlabs NTP module
#   which interfaces will accept connections
ntp::interfaces:
  - '127.0.0.1'
  - '192.168.250.10'
#   which nodes can connect
ntp::restrict:
  - 'default kod nomodify notrap nopeer noquery'
  - '-6 default kod nomodify notrap nopeer noquery'
  - '127.0.0.1'
  - '-6 ::1'
  - '192.168.250.0/24'
  - '-6 fe80::'
```

Now that we've defined the input data, let's re-run this command to effect our changes.

```
[vagrant@client ~]$ sudo puppet apply --environment test --execute 'include ntp'
Notice: Compiled catalog for client.example.com in environment test in 0.56 seconds
Notice: /Stage[main]/Ntp::Config/File[/etc/ntp.conf]/content: content changed '{md5}c9d83653966c1e
Notice: /Stage[main]/Ntp::Service/Service[ntp]: Triggered 'refresh' from 1 events
Notice: Finished catalog run in 0.33 seconds
```

As you can see, Puppet has updated the configuration file and restarted the service. You can validate the changes by examining the revised */etc/ntp.conf*.

```
[vagrant@client ~]$ grep 192.168.250 /etc/ntp.conf
restrict 192.168.250.0/24
interface listen 192.168.250.10
```

Executing Multiple Modules with Hiera

The modern, best-practice way to assign modules to a node's class list is to define the classes within Hiera. This takes advantage of the Hiera hierarchy to customize the load list. As you might recall, the hiera hierarchy we defined earlier includes the environment within the hieradata load path.

Let's go ahead and set that up now. First, edit the following file:

```
[vagrant@client ~]$ $EDITOR /etc/puppetlabs/code/environments/test/manifests/site.pp
```

In this file we're going to tell Puppet to load every node's class list from Hiera. Add the following to the file:

```
hiera_include('classes')
```

Let's go ahead and add the same to */etc/puppetlabs/code/environments/production/manifests/site.pp* while we are at it.

At this point you need to assign the class list in Hiera. Let's go ahead and open up the *global.yaml* file in our environment's hieradata directory.

```
[vagrant@client ~]$ $EDITOR /etc/puppetlabs/code/environments/test/hieradata/global.yaml
```

Within this file create a top-level array named `classes`.

```
---
classes:
  - 'ntp'
```

Now that the ntp module is listed in every node's class list, you can test the module without supplying `--execute` on the command line.

```
[vagrant@client ~]$ puppet apply --environment test
Notice: Compiled catalog for client.example.com in environment test in 0.61 seconds
Notice: Finished catalog run in 0.14 seconds
```

> Wait, what happened? This time it didn't do anything. That's because the module is properly **Idempotent**. The configuration hasn't changed and the service is still running, therefore no changes were necessary.

Let's give the NTP module something to do by stopping the service.

```
[vagrant@client ~]$ sudo service ntpd stop
Shutting down ntpd:                                        [  OK  ]
[vagrant@client ~]$ puppet apply --environment test /etc/puppetlabs/code/environments/test/manifes
Notice: Compiled catalog for client.example.com in environment test in 0.71 seconds
Notice: /Stage[main]/Ntp::Service/Service[ntp]/ensure: ensure changed 'stopped' to 'running'
Notice: Finished catalog run in 0.18 seconds
```

You may find older documentation that suggests defining the node's data directly within the *site.pp* file. In large environments this quickly devolves into a maintenance nightmare of node assignments and inheritence. When combined with altering the modules available in different environments, managing module assignment consistently went from impractical to implausible.

Examining a Module

Regardless of the quality score or community rating given to a module, it is always best to examine modules carefully to ensure they will work well in your environment. Badly written modules can create unintended chaos when put into use within your existing module space.

Here is a short list of things you should examine the code base to determine if the module will work well for you.

- OS Support: May not support your operating system properly.
- Module Namespace: May require dependency modules named the same as modules you use.
- Environment Assumptions: May enforce local assumptions that won't work in your environment.
- Sloppy code: Could require or overwrite global variables.
- Resource Namespace: Could use simple resource names which conflict with others used already.
- Greedy collectors: Could utilize collectors which accidentally grab unrelated resources.

You should utilize the above list of things to check for when building your own modules.

Reviewing Modules

In this chapter we have gone through the installation, configuration, testing, and execution of a module from the Puppet Forge.

We setup the directory environments *test* and *production*. We have installed the NTP module and its dependent StdLib into the modules directory of the test environment. We have configured the NTP module using Hiera configuration data based on the instructions from the modules web page. We have watched as the module operated to configure and run the NTP service on our system.

CHAPTER 13

Designing a Custom Module

In this chapter we will go through the process of creating a custom module. We will cover how to create each component of a module, including:

- Classes and subclasses within the module
- Files distributed by the module intact
- Templates used to create customized files
- Definitions which can be re-used for processing multiple items
- Facts which are added to each node
- Pure ruby functions which provide features unavailable in the Puppet configuration language
- Tests which validate that the module works as expected

Let's get started.

Choosing a Module Name

A module can be named anything which begins with a letter and contains only lowercase alphanumeric characters and underscores. A hyphen is not allowed within a module name. Here are some valid and invalid module names.

mymodule	valid
3files	invalid
busy_people	valid
busy-people	invalid

It is important to choose your module name carefully to avoid conflicts with other modules. As each module creates its own namespace, only one module may exist

within a Puppet catalog with a given name at the same time. For most intents and purposes, this means that you can never use two modules with the same name.

When naming modules, I try to avoid naming the module anything which conflicts with a module name available in the Puppet Forge. The reason for this is that I never know when another team might utilize that module for their project, or when a module I download might use the Puppet Forge module as a dependency.

It's possible to manage conflicting module names by utilizing them within separate environments. We'll cover how to do this in the Customizing Environments chapter.

Avoiding Reserved Names

There are namespaces which are internal to and used by Puppet itself. For this reason you can never create a module with the following names:

main The `main` class contains any resources not contained by any other class.

settings The `settings` namespace contains variables with the Puppet configuration settings.

trusted The `trusted` namespace contains facts taken from the client's certificate, as validated by a Puppet server.

Generating a Module Skeleton

Puppet will generate an empty skeleton for you with the `puppet module generate` command. As mentioned at the start of the book, your first module will be made to manage the Puppet agent itself. Let's create that now. Use your own name or organization name instead of *myorg* below.

When you generate a module, it will ask you a number of questions. Default values used if you just hit `ENTER` are included in brackets after each question.

```
$ puppet module generate myorg-puppet
We need to create a metadata.json file for this module.  Please answer the
following questions; if the question is not applicable to this module, feel free
to leave it blank.

Puppet uses Semantic Versioning (semver.org) to version modules.
What version is this module?  [0.1.0]
-->
```

After you have answered all of the questions, it will generate the module in your current directory.

```
Notice: Generating module at /home/vagrant/myorg-puppet...
Notice: Populating templates...
```

```
Finished; module generated in myorg-puppet.
myorg-puppet/manifests
myorg-puppet/manifests/init.pp
myorg-puppet/spec
myorg-puppet/spec/classes
myorg-puppet/spec/classes/init_spec.rb
myorg-puppet/spec/spec_helper.rb
myorg-puppet/tests
myorg-puppet/tests/init.pp
myorg-puppet/Gemfile
myorg-puppet/Rakefile
myorg-puppet/README.md
myorg-puppet/metadata.json
```

If you prefer to skip the questions and edit the file yourself, add `--skip-interview` to the command line.

Modifying the Default Skeleton

This topic isn't important right now, but after you've been creating modules for some time you may to tune the default module skeleton to include things you want in your modules. This is done by placing your revised skeleton in the *~/.puppetlabs/opt/puppet/cache/puppet-module/skeleton* directory.

Previous versions of Puppet used the *~/.puppet/var/puppet-module/skeleton* directory. You'll need to copy your skeleton to the new directory.

```
$ cp -r ~/.puppet/var/puppet-module/skeleton ~/.puppetlabs/opt/puppet/cache/puppet-module/
```

There are skeletons that include better test suites, and skeletons that include common examples for different application environments. You can find Puppet module skeletons by searching for "puppet skeleton" on GitHub.

You can install multiple skeletons in a directory of your choice, and select one for the module you will be building like so:

```
$ puppet module --module_skeleton_dir=~/skels/passenger-app generate myorg-railsapp
```

Understanding Module Structure

Let's review the files and directories created in your new module. Each of the following are files are fixed, unchangeable paths built into Puppet's expectations and utilization of modules.

manifests/	Directory where code manifests (classes) are read.
files/	Directory containing files served by your module.
templates/	Directory containing templates parsed for custom files.
lib/	Directory containing ruby functions or facts used.
specs/	Directory containing unit tests to validate the manifests.
tests/	Directory containing test invocations of manifests.
facts.d/	Directory containing external facts to be distributed.
metadata.json	File containing meta information about the module.

Creating a Class Manifest

Let's start within the *manifests/* directory of your module. In this directory you will find a single file named *init.pp*. This file must exist within every Puppet module, and it must contain the definition for the *base class*. The base class is a class with the same name as the module. Let's take a look at that now.

```
[vagrant@client ~]$ cd myorg-puppet/manifests
[vagrant@client manifests]$ cat init.pp
class puppet {

}
```

Right now this class has no definition. That is an acceptable situation--the class must be defined, but it doesn't have to do anything. We'll flesh it out in the very next section, after we discuss the difference between a class and a manifest.

You will observe that the file contains a documentation template above this class definition. Go ahead and ignore this for now. We'll cover documentation after we have covered each of these pieces that the docs will refer to, in Documenting the Module.

What is a Class?

At this point in the book you have created and used manifests. In review, manifests:

- Use Puppet configuration language to define configuration policy
- Contain Resources which describe how their target type should be configured
- Execute immediately with the `puppet apply` command

The good news for you is that a *Class* is a manifest with special properties. It uses Puppet configuration language to declare resources exactly the same as done in a manifest. This means that if you know how to write a manifest, then you know how to write a class.

Before we get started, let's review quickly the special properties that make a class different from a manifest. A class is different in the following ways:

- A Class is a manifest that can be called by name
- A Class has a *namespace* or variable scope within its name
- A Class is not used until called by name
- A Class may include or be included by other modules
- A Class may be passed parameters when called

Accepting Input

Let's define parameters your Puppet module will accept. Adjust the *init.pp* file to look something like this.

```
class puppet(
  # input parameters and default values for the class
  $version = 'latest',
  $status  = 'running',
  $enabled,              # required parameter
) {

  # This is where you place your resources.

}
```

Here we have declared three input parameters. Two of them have default values, which will only be used if input is not provided when the class is called. The third value `enabled` is required, and will generate an error if a value is not provided.

Classes get their input parameters from four possible places. If the data is not available in the first place, it looks in the next according to the following flow:

1. Parameters can be explicitly passed using a resource definition in a manifest or an ENC.
2. Parameters can be supplied by Hiera data.
3. Parameters can be supplied within the Class manifest.

Modern Puppet practice is to define all class input parameters within Hiera. Using resource definitions for invoking classes was common prior to Hiera, but is no longer recommended for normal use.

We'll discuss how to use ENCs in Part IV: Advanced Puppet.

Validating Input with Types

Puppet 4 has a new type system which validates input parameters. This improves code quality and improves readability of the code.

In older versions of Puppet it was common to perform input validation like this:

```
class puppet(
  # input parameters and default values for the class
  $version = 'latest',
  $status  = 'running',
  $enabled,
) {
  validate_string( $version )
  validate_string( $status )
  validate_bool( $enabled )
...resources defined below...
```

While this looks easy with three variables, it could consume pages of text with a lot of input variables. It was fairly common for the first resource defined in a manifest to be down below line 180.

Puppet 4 allows you to define the type when declaring the parameter now, which both shortens the code and improves the readability. When declaring the parameter, simply add the type before the variable name. This declares the parameter and add validation for the input on a single line.

I think you'll agree the following is significantly more readable:

```
class puppet(
  # input parameters and default values for the class
  String $version = 'latest',
  Enum['running','stopped'] $status  = 'running',
  Boolean $enabled,
) {
...class resources...
}
```

It is not necessary for you to declare a type. If you are passing in something which can contain multiple types of data, simply leave the definition without a type as shown in the previous section. A parameter without an explicit type defaults to a type named `Any`.

Best Practice: Use explicit data types for all input parameters. Avoid accepting ambiguous values which must be introspected before use.

You can (and should) also declare types for lambda block parameters:

```
split( $facts['interfaces'] ).each |String $interface| {
  ...lambda block...
}
```

Valid Types

The type system is hierarchial, where you can allow multiple types to match by defining at a higher level in the tree. Let's examine this now.

- *Scalar* - accepts any of:
  - *Boolean* - true or false
  - *String* - ASCII and UTF8 characters
    - *Enum* - a specified list of Strings
    - *Pattern* - subset of Strings which match a given Regular expression
  - *Numeric* - accepts either:
    - *Float* - numbers with periods; fractions
    - *Integer* - whole numbers without periods
  - *Regexp* - a Regular Expression[1]
- *Collection* - accepts any of:
  - *Array* - list containing other data types
  - *Tuple* - an Array with specific types in specified positions.
  - *Hash* - key/value associated pairs. Keys are Strings, Values can be any type.
  - *Struct* - a Hash with specified types for the key and value pairs.

All of the above are part of a type named *Data*. You can use Data to indicate that you accept any of these types. There are a few special types that don't fall under Data:

- *Variant* - a special type which matches values of a specified list of types.
- *Optional* - a special type which accepts an optional value of a specific type.
- *Undef* - a special type which only accepts undefined values.
- *Callable* - a special type which refers to a lambda block.
- *CatalogType* - a native Puppet resource type, such as *Class*, *Node*, etc.
- *Runtime* - contains references to objects available during runtime, e.g. Ruby class names.

All of these types are considered type *Any*, also refered to as *Object* in some error messages. There's no point in declaring a type of Object because you'd be saying that it

1 You can learn more about Regular Expressions from Mastering Regular Expressions

accepts anything, but it can be useful to help you understand when you see error messages like this:

```
Error: Evaluation Error: Error while evaluating a Function Call, function 'alarm' called with mis-
expected:
  alarm(Array[String] input_strings) - arg count {1}
actual:
  alarm(Tuple[Integer, Float, Tuple[Integer, String, Float], String]) - arg count {1} at /vagrant/
type mismatch, an Array[Object] cannot be used where an Array[Integer] is expected
```

Best Practice: Even when values may be multiple data types, most circumstances you are expecting a `Scalar` data type, rather than one of the internal objects. Use `Scalar` for parameters where the values can be multiple types.

Accepting Values

The type system allows for validation of not only the parameter type, but of the values within structured data types. For example:

```
Integer[13,19]           # teenage years
Array[Float]             # an array containing only Floating point numbers
Array[ Array[String] ]   # an array containing only Arrays of Strings
Array[ Numeric[-10,10] ] # an array of Integers or Floats between -10 and 10
```

For structured data types, the range parameters indicate the size of the structured type, rather than a value comparison. You can check both the type and size of a Collection (Array or Hash) using two values: the type and an array with the size parameters.

```
Array[String,0,10]      # an array of 0 to 10 string values
Hash[String,Any,2,4] ]  # a hash with 2 to 4 pairs of string keys with any value
```

To get even more specific about key and value types for a Hash, a `Struct` specifies exactly which data types are valid for the keys and values.

```
# This is a Hash with short string keys and floating point values
Struct[{
  day => String[1,8],       # keys are 1-8 characters in length
  temp => Float[-100,100],  # values are floating point Celcius
}]

# This is a Hash which can contain only 3 keys named after ducks
Struct[{
  duck  => enum['Huey','Dewey','Louie'],
  loved => Boolean,
}]
```

Like what a Struct does for a Hash, you can specify which types of data for an Array using the `Tuple` type. The Tuple can list specific types in specific positions, along with a specified minimum and optional maximum number of entries.

```
# an array with three integers followed by one string (explicit)
Tuple[Integer,Integer,Integer,String]

# an array with 2 Integers followed by 0-2 Strings (length 2-4)
Tuple[Integer,Integer,String,2,4]

# an array with 1-3 Integers, and 0-5 Strings
Tuple[Integer,Integer,Integer,String,1,8]
```

That last one is pretty hard to understand so let's break it down:

- Integer in position 1 (required minimum)
- Integers in positions 2-3 are optional (above minimum length)
- String in position 4 is optional (above minimum length)
- The final type (String) may be repeated up to maximum 8

As you can see, the ability to be specific for many positions in an Array makes `Tuple` a powerful type for well-structured data.

The *Variant* and *Optional* types allows you to specify valid alternatives.

```
Variant[Integer,Float]         # the same as Numeric type
Variant[String,Array[String]]  # a String or an array of Strings
Variant[String,undef]          # a String or nothing
Optional[String]               # same as the previous line
Optional[String,Integer]       # String, Integer, or nada
```

You can also check the size of the value of a given type:

```
String[12]           # a String at least 12 characters long
String[1,40]         # a String between 1 and 40 characters long
Array[Integer,3]     # an Array of at least 3 integers
Array[String,1,5]    # an Array with 1 to 5 strings
```

You can use all of these together in combination:

```
# An arrray of values which are thumbs up or thumbs down
Array[ Enum['thumbsup','thumbsdown'] ]

# An arrray of values which are thumbs up, thumbs down, or a value from 1 to 5
Array[ Variant[ Integer[1,5], Enum['thumbsup','thumbsdown'] ] ]
```

Testing Values

In addition to defining the type of parameters passed into a class, defined type, or lambda, you can perform explicit tests against values in a manifest. Use the =~ operator to compare a value against a type to determine if the value matches the type decla-

ration. For instance, if a value could be one of several types, you could determine the exact type so as to process it correctly.

```
class myclass(
  # accepts on input value of either Integer or String
  Variant[Integer,String] $input_value,
) {

  if( $input_value =~ String ) {
    notice( "Received string ${input_value}" )
  }
  elsif( $input_value =~ Integer ) {
    notice( "Received integer ${input_value}" )
  }
}
```

You can also determine if a type is available within a Collection with the `in` operator.

```
if( String in $array_of_values ) {
  notice('Found a string in the list of values.')
}
else {
  notice('No strings found in the list of values.')
}
```

The `with` function can be useful for type checking as well:

```
with($password) |String[12] $secret| {
  notice( "The secret '${secret}' is a sufficently long password." )
}
```

You can likewise test value types using `case` and `selector` expressions:

```
case $input_value: {
  Integer: { notice('Input plus ten equals ' + ($input_value+10) ) }
  String: { notice('Input was a string, unable to add ten.') }
}
```

You can test against *Variant* and *Optional* types as well

```
if( $input =~ Variant[ String, Array[String] ] ) {
  notice( 'Values are strings.' )
}
if( $input =~ Optional[Integer] ) {
  notice( 'Values is a whole number or undefined.' )
}
```

A type compares successfully against its exact type, and parents of it's type, so the following statements will all return true:

```
  'text' =~ String  # exact
  'text' =~ Scalar  # Strings are children of Scalar type
  'text' =~ Data    # Scalars are a valid Data type
  'text' =~ Object  # all types are Objects
```

If you don't like the default messages displayed when compilation fails due to a type mismatch, you can customize your own by using the `assert_type` function with a lambda block.

```
assert_type(String[12], $password) |$expected, $actual| {
  fail "Passwords shorter than 12 characters can be cracked easily. You supplied: ${actual}"
}
```

Matching Regular Expressions

You can evaluate Strings against regular expressions to determine if they match using the same `=~` operator. For instance, if you are evaluating filenames to determine which ones are Puppet manifests, the follow example would be useful:

```
$manifests = $filenames.filter do |$filename| {
  $filename =~ Regexp[/\.pp$/]
}
```

You can match against multiple exact strings and regular expressions with the `Pattern` type:

```
$placeholder_names = $victims.filter do |$name| {
  $name =~ Pattern['alice','bob','eve','(?i)^j.* doe',/(?i)^j.* roe/]
}
```

Only the `Regexp` type can use variable interpolation, placing another variable within the match string:

```
  $fullname =~ Regexp["${first_name} [\w\-]+"]
}
```

Declaring Resources

As we mentioned previously, Class manifests are almost exactly like the manifests you used with `puppet apply`. To that end, I am sure that you remember (or can look back in the book) the resources desclared to install a package and start a service. Go ahead and fill in the class manifest with those resources right now.

When you are done, your manifest will look something like this:

```
class puppet(
  String $version = 'latest',
  Enum['running','stopped'] $status  = 'running',
  Boolean $enabled,
) {
  # Install the Puppet agent
  package { 'puppet-agent':
```

```
    version => $version,
    notify  => Service['puppet'],
  }

  # Manage the Puppet service
  service { 'puppet':
    ensure    => $status,
    enable    => $enabled,
    subscribe => Package['puppet-agent'],
  }
}
```

This is very similar to the package and service manifest built out earlier in the book. As per best practice, we have defined dependencies to ensure that the service is restarted if the package is updated.

Using Hiera Data

At this point, you might be asking yourself why we didn't use Hiera calls to get the input values for the class. As discussed in the Separating Data from Code chapter, Hiera provides a configurable, hierachial mechanism to provide input data for manifests.

One of the special properties of Class manifests is that you don't need to use the explicit `hiera()` function calls shown in that chapter. Instead, you define the hiera values using the module's namespace. Let's go ahead and define those values now in */etc/puppetlabs/code/hieradata/global.yaml*.

```
# /etc/puppetlabs/code/hieradata/global.yaml
puppet::version = 'latest'
puppet::status  = 'stopped'
puppet::enabled = false
```

Without any function calls, these variables will be provided to the puppet module as parameter input, overriding the default values provided in the class declaration.

You cannot define input parameters as hash keys for the module name.

Given that Hiera uses the `::` separator to define the data hierarchy, you might think that it would be easier to define the input parameters as hash entries underneath the module. And yes, I agree that the following looks very clean:

```
puppet:
  version = 'latest'
```

```
  status  = 'stopped'
  enabled = false
```

Unfortunately, it does not work. The key for an input parameter must match the complete string of the module namespace plus the parameter name. So the file must be written using the definition shown before the warning above.

Sharing Files

You are probably thinking to yourself that this manifest isn't sufficient. One can't generally install and run software without configuring it. So it's time to add a configuration file.

The first step is to create a directory in which to store files. Puppet has a built in map for a module's shared files. They must reside in a *files* directory within the module directory.

```
$ cd myorg-puppet
$ mkdir files
```

Now let's create a small configuration file in that directory to test out the concept.

```
$ $EDITOR files/puppet.conf
```

For now, let's make a very simple change to Puppet's configuration. For a reason we'll discuss in the next section, let's make this a very simple change. Let's make the logging a bit more verbose.

```
[main]

# This is used for "puppet apply"
[user]
        log_level = info
```

Now we shall add a file resource to the class manifest. Unlike how we used it in the manifest, instead of using `content` attribute, we'll use a `source` attribute. Files specified by this attribute are copies exactly as they are. Type the `source` exactly as shown below:

```
file { '/etc/puppetlabs/puppet/puppet.conf':
  ensure => file,
  owner  => 'root',
  group  => 'wheel',
  mode   => '0644',
  source => 'puppet:///modules/puppet/puppet.conf',
}
```

The target of a source directive can be another local file specified by a fully qualified path, or it can be a URI to a file from the module. The most common specification is to use three slashes in a row `///` which leaves the server field in the URI blank. This

means to get the file from the Puppet server the agent is currently talking to. The three slashes URI also works seamlessly with `puppet apply` on the local system.

The path */modules/puppet* is a special path which maps to the *files* directory in the puppet module. You can place subdirectories within the files directory if you have many files you need to organize.

Best Practice: Avoid specifying an explicit Puppet server in the URI source for a file, as the module will fail if used in an environment which doesn't have direct access to that exact server. Instead, synchronize the dependent files to all Puppet servers which serve the module that specifies the files.

You can specify an array of file sources. Puppet will go through the array of sources and use the first file it finds.

We won't use it for this module, but it's possible to synchronize all files in a directory. With the optional `recurse` and `purge` parameters it is possible to synchronize a directory tree of files.

```
file { "/etc/program/":
  ensure  => directory,
  owner   => 'root',
  group   => 'wheel',
  mode    => '0444',
  recurse => true,   # go into subdirectories
  replace => true,   # replace any files that already exist
  purge   => false,  # don't remove files that we don't have
  links   => follow, # follow symbolic links and modify the link target
  force   => false,
  source  => 'puppet:///modules/puppet/puppet.conf',
}
```

As you can see above, there are many parameters for controlling how files are installed, and under what situations they will replace other files. You can find detailed explanations for these attributes at Puppet 4 Type Reference: File.

In my experience Puppet is not well suited for synchronizing very large directories, or very large files. I have used it very successfully for small trees of HTML, XML, JSON, or INI configuration files, such as those used by Java application services. When synchronizing very large directories or files nearing or greater than a gigabyte in size it is better to synchronize files using utilities designed for those kinds of transfers, e.g. `rsync`, `wget --mirror`, and for some purposes `git` can be significantly higher performance.

It is possible to source files which are not within a module from the Puppet server. However, this practice is deprecated and not recommended for many good reasons, so I am not going to cover that here.

Best Practice: Always places files in the same module as the manifests which use the files.

Parsing Templates

I can imagine you are saying to yourself, "Not every system will get the same configuration." Good point! Let's build a customized template for this configuration file.

First, create a directory in which to store templates. Like files, templates have their own directory in the module structure.

```
$ cd myorg-puppet
$ mkdir templates
```

The template we are going to build will utilize four variables. Let's build a manifest that supplies those variables. First, let's modify the module input to set default values for each of these variables:

```
class puppet(
  String $version = 'latest',
  Enum['running','stopped'] $status  = 'running',
  Boolean $enabled,
  String loglevel = 'warning',
  String agent_loglevel = 'warning',
  String apply_loglevel = 'warning',
  String server = 'puppet.example.net',
) {
```

Instead of String it would be more specific to use a type of `Enum['debug','info','notice','war ning','err','alert','emerg','crit','verbose']` for each of the loglevels. I didn't do this only for page formatting/display reasons.

With this declaration, we now have seven variables in our class's scope which we can use in a template.

There are two different template parsers within Puppet:

- The new Puppet EPP templates which use Puppet variables and Puppet functions.
- The well-known Ruby ERP templates which use Ruby language and functions.

It doesn't matter which one you use, so we'll teach you to use each of these.

Common Syntax

Both Puppet EPP and Ruby ERB files are plain text files which contain some common syntax and template tags. Outside of these tags is normal, unprocessed text. Within the tags are Puppet or Ruby variables and code.

<%= variable or code %>	This tag is replaced with the value of the variable, or result of the code.
<% code %>	The code is executed but no result is returned (unless the code calls an output function)
<%# comment for an editor of the file or module %>	This tag is removed from the output.
<% code block -%>	Trailing whitespace including newlines are suppressed. Useful to avoid blank lines in output.
<%- code block %>	Leading whitespace including a newline is suppressed. Useful when code is indented for readability.
<%= variable or code -%>	Result of the code with trailing whitespace removed. (Trim of leading whitespace is not available)
<%% -or- %%gt;	Doubled percent signs are replaced with single percent sign without any code evaluation.

Don't worry about memorizing these patterns just yet. We will use these tags in the next few pages, and you'll get to see their use in context.

Using Puppet EPP Templates

First let's adjust the file declaration from the previous section. We'll remove the `source` attribute and replace it with a `content` attribute. The content will be provided by the `epp()` function.

```
file { '/etc/puppetlabs/puppet.conf':
  ensure  => ensure,
  owner   => 'root',
  group   => 'wheel',
  mode    => '0644',
  content => epp('puppet:///puppet/puppet.conf.epp'),
}
```

The `epp()` function takes a two arguments:

URI of the Puppet EPP template
: The format of that URI is always `puppet:///modulename/filename.epp`. The file should be placed in the *templates* directory of the module. Puppet EPP templates end with the *.epp* extension to indicate that the file contains tags for the Puppet EPP template processor.

A hash of parameters for input
: An optional hash of input parameters for the template, described in **Providing Parameters** below.

Let's create a template file.

```
$ cd myorg-puppet/templates
$ $EDITOR puppet.conf.epp
```

The template should look something like this:

```
# Generated by Puppet EPP template processor
[main]
  log_level = <%= $loglevel %>

# This is used by "puppet agent"
[agent]
  log_level = <%= $agent_loglevel %>
  server = <%= $server -%>.example.net

# This is used for "puppet apply"
[user]
  log_level = <%= $apply_loglevel %>
```

This example utilizes four variables. Each instance of `<%= $variable %>` is replaced with the Puppet variable from the current scope. We added those variables to the manifest as our very first step.

In this declaration, the `epp()` function processes the EPP template and provides the content to be placed in the file. It does this by replacing the variable lookups in the file we created with variables from the current variable scope (within this class).

Go ahead and test this change right now with `puppet apply`. You will see the contents of the puppet configuration file get updated.

You can lookup variables from another class using the module's scope, the same as you would inside a Puppet template. For example, if we wanted to use the same loglevel as used by MCollective, the following would work.

```
loglevel = <%= $mcollective::loglevel -%>
```

EPP templates can do far more than variable replacement. You can put any Puppet function within `<% ... %>` tags without the equals sign. Here's an example that uses conditional evaluation to limit duplicate assignment of loglevels which don't differ.

```
[user]
<% if $apply_loglevel != $loglevel -%>
  log_level = <%= @apply_loglevel %>
<% end -%>
```

By placing this line of output within the Puppet if condition, the line will only be output if the condition matches. This will avoid outputting the configuration line if the loglevel matches the main loglevel, thus simplifying the configuration file.

Providing Parameters

You can optionally provide a hash of input parameters for the template. This is very useful when the Puppet variable names don't match the variable names used in the template. For example:

```
content => epp('puppet:///puppet/puppet.conf.epp',
  { String server => $server, String loglevel => $loglevel }
),
```

When providing parameter input, then the very first line of the template should contain the following special syntax used for accepting the variables.

```
<%- |$server, $loglevel = 'warning'| -%>
```

This input format exactly matches the input assignments used for class parameters, where the `server` value is required to be provided, while the `loglevel` has a default value of *warning* if no value is provided.

These two forms must match each other, where all required parameters must be supplied for templates which define them.

Best Practice: Always place the parameters you will use in the template at the top, and send them explicitly when declaring the template file in your manifest. This ensures the greatest readability, allowing a user to avoid searching multiple files to determine where a value was sourced from.

Iterating over Values

You can use any Puppet function within the template tags. By far the most common operation to perform within a template is to iterate through some values.

Here's an example where we use the `each()` function to iterate through an array of tags which should be used to limit which resources are applied to the node, as we discussed in Part I. This example uses the dash creatively to suppress linefeeds and output multiple Puppet servers on a single line:

```
[agent]
  tags = <% $taglist.each do |$tagname| -%>
<%= $tagname + ',' -%>
```

```
<% end -%>

# an extra linefeed to terminate the line above
```

This function is written exactly the same as the iteration examples shown to you in Using Puppet Configuration Language section on Looping Through Iterations.

Learning More

Complete documentation for EPP templates is available at Puppet Functions: epp.

Using Ruby ERB Templates

First, let's adjust the file declaration from the previous section. We'll remove the `source` attribute and replace it with a `content` attribute.

```
file { '/etc/puppetlabs/puppet.conf':
  ensure  => ensure,
  owner   => 'root',
  group   => 'wheel',
  mode    => '0644',
  content => template('puppet:///puppet/puppet.conf.erb'),
}
```

The `template()` function takes a single argument: the URI of the ERB template. The format of that URI is always `puppet:///`*`modulename`*`/`*`filename`*. The file should be placed in the *templates* directory of the module. ERB templates should end with the *.erb* extension to indicate that the file contains tags for the ERB template processor.

Let's create the ERB template file.

```
$ cd myorg-puppet/templates
$ $EDITOR templates/puppet.conf.erb
```

The contents of the file should look like this:

```
# Generated by Puppet ERB template processor
[main]
  log_level = <%= @loglevel %>

# This is used by "puppet agent"
[agent]
  log_level = <%= @agent_loglevel %>
  server = <%= @server -%>.example.net

# This is used for "puppet apply"
[user]
  log_level = <%= @apply_loglevel %>
```

Just as our EPP example, this simple example uses four variables. Each instance of `<%= @variable %>` is replaced with the value of the Puppet variable named after the `@` sign. The variables named with the `@` sign must exist in the same scope (within the module class) as the template declaration.

There are many other things you can do within an ERB template. You can lookup variables from another class using the `scope.lookupvar()` function, or use `scope[]` as if it was a Hash. For example, if we wanted to use the same loglevel as used by MCollective, either of the following would work.

```
loglevel = <%= scope.lookupvar('mcollective::loglevel') -%>
loglevel = <%= scope['::mcollective::loglevel'] -%>
```

You can call Puppet functions using `scope.function_`*`puppet_function`*`()`. For example, you could call the Hiera function to lookup Hiera values within templates (although this practice is strongly discouraged). This would be done by using `scope.function_hiera()` to call the same `heira()` function we used when introducing Hiera.

```
server = <%= scope.function_hiera( ['puppet::server'] ) -%>
```

Best Practice: Avoid placing direct Hiera calls within the template, as it divides the source of data for the template between the manifest file and hiera, ensuring confusion. Instead, source the Hiera variables within the manifest so that all variables are within scope.

As ERB templates were intended for inline Ruby development, you can put any Ruby statement within `<% ... %>` tags without the equals sign. Here's an example that would limit duplicate assignment of loglevels which don't differ.

```
[user]
<% if @apply_loglevel != @loglevel -%>
  log_level = <%= @apply_loglevel %>
<% end -%>
```

By wrapping this line of the template within the Ruby block, it will skip outputting the configuration line if the loglevel matches the main loglevel, thus simplifying the configuration file.

Go ahead and test this change right now with `puppet apply`. You will see the contents of the puppet configuration file get updated.

Iterating over Values

Here's an example where we use the Ruby `each()` function to iterate through an array of tags which should be used to limit which resources are applied to the node, as we

discussed in Part I. This example uses the dash creatively to suppress linefeeds and output multiple Puppet servers on a single line:

```
[agent]
  tags = <% @taglist.each do |tagname| -%>
<%= tagname + ',' -%>
<% end -%>
```

You'll note that we don't put an `@` sign before the variable name. That is because we are not referencing a variable in the Puppet module class, but instead from the local loop shown in this example.

Learning More

More documentation for using ERB syntax with Puppet can be found at Using Puppet Templates. ERB is commonly used by Ruby in many other situations, so you can find advice and help for using ERB syntax with any search engine.

Creating Readable Templates

You may have noticed that both the original template and this block sometimes utilize a leading dash in the closure `-%>`. The dash tells the interpretter to supress the following linefeed, thus avoiding blank lines in the file output. This is commonly used to keep comments or Ruby statements from adding blank lines to a line, but can also be used to concatenate two sequential lines.

You want to avoid trimming whitespace with the dash if that is the end of the line, or two lines will be joined together.

Best Practice: Use Puppet EPP templates with Puppet Configuration Language in both the manifests and the templates. Specify parameters for the template explicitly for clarity and readability.

I cannot personally express the preceding suggestion strongly enough. I cannot tell you how many hours I have spent searching through dozens of manifests to try and determine where a variable used in a template was sourced from.

Building Subclasses

When building a module you may find yourself with several different related components, some of which may not be utilized on every system. For example, our Puppet class should be able to configure both the Puppet agent and a Puppet server. In situations like this, it is best to break your module up with subclasses.

Each subclass is named within the scope of the parent class. For example, the class which configures the Puppet agent would make sense to name `puppet::agent`.

Each subclass should be a separate manifest file, stored in the *manifests* directory of the module, and named for the subclass followed by the *.pp* extension. For example, our Puppet module could be expanded to have the following classes:

Class Name	File Name
puppet	manifests/init.pp
puppet::agent	manifests/agent.pp
puppet::server	manifests/server.pp

As our module currently only configures the Puppet Agent, let's go ahead and move all resources from the `puppet` class into the `puppet::agent` class. When we are done the files might look like this:

```
# manifests/init.pp
class puppet(
  # common variables for all Puppet classes
  String $version  = 'latest',
  String $loglevel = 'warning',
) {
  # no resources in this class
}

# manifests/agent.pp
class puppet::agent(
  # input parameters specific to agent subclass
   Enum['running','stopped'] $status  = 'running',
  Boolean $enabled,             # required parameter
)
inherits puppet {

  all of the resources previously defined
}

# manifests/server.pp
class puppet::server() {
  # we'll write this in Part III of the book
}
```

Best Practice: Any time you would need an `if/then` block in a module to handle different needs for different nodes, use subclasses instead for improved readability.

One last change will be to adjust Hiera to reflect the revised class name:

```
# /etc/puppetlabs/code/hieradata/global.yaml
classes:
        puppet::agent

puppet::loglevel = 'info'
puppet::agent::version = 'latest'
puppet::agent::status  = 'stopped'
puppet::agent::enabled = false
```

With these small changes we have now made it possible for a node to have the Puppet agent configured, or the Puppet server configured, or both.

Remember that module parameters must be supplied with the entire module class (e.g. `puppet::agent` plus `::` and the variable name. You cannot define parameters as hash keys under the module name.

Understanding Variable Scope

Modules may only declare variables within the module's namespace (also called *scope*). This is very important to remember when using subclasses within a module, as each subclass has its own scope.

```
class puppet::agent {
   # these two definitions are the same
   $version = '1.0.1'
   $::puppet::agent::version = '1.0.1'
```

A module may not create variables within the top scope or another module's scope. Any of the following declarations will cause a compilation error:

```
class puppet {
  # FAIL: can't declare top-level variables
  $::version = '1.0.1'

  # FAIL: Can't declare variables in another class
  $::mcollective::version = '1.0.1'

  # FAIL: no, not even in the parent class
  $::puppet::version = '1.0.1'
```

While you cannot change variables in other scopes, you can use them within the current scope.

```
    notify( $variable )             # variable in current, parent, node, or top scope
    notify( $::puppet::variable )   # variable in the parent scope
    notify( $::variable )           # variable in the node or top scope
```

The first invocation could return a value from an in-scope variable, a variable from the parent scope, or a top-scope variable. A person would have to search the module to be certain a local scope variable wasn't defined. Furthermore, a declaration added to the manifest above this could assign a value different from what you intended to use. The latter form is explicit and clear about the source.

Best Practice: Always refer to out of scope variables with the explicit `$::` prefix for clarity.

The one and only time where you should refer to an out of scope variable without the `$::` prefix is when redeclaring the variable within the current scope.

```
$sumtotal = 30   # overrides the higher scope variable within this class
$sumtotal += 10  # overrides the higher scope variable with a local addition of 10
```

The latter form looks like a redefinition, which as you might recall is not possible within Puppet. However, the `+=` operator actually creates a variable in the local scope with a value of the higher scope variable plus the value specified.

Reusing Defined Types

Puppet classes are what's known as *singletons*. No matter how many places they are called with `include` or `require` functions, only one copy of the class exists in memory. Only one set of parameters are used, and only one set of scoped variables exist.

There will be times that you may want to invoke Puppet resources multiple times with different input each time. For that purpose you create a *defined type*.

Defined types are manifests that look almost exactly like subclasses:

- They are placed in the *manifests/* directory of a module.
- They are named within the module namespace exactly like subclasses.
- The file name should be named for the defined type, and end with the *.pp* suffix.
- They begin with parenthesis that define parameters which are accepted.

Unlike classes, defined types can be called over and over again. This makes them suitable for use within the lamba of an iterator. We'll use an iterator in our puppet class in the next section. To demonstrate the idea now, here's an example from a `users` class.

```
# modules/users/manifests/create.pp
define users::create(
  Integer $uid,
  Optional[Integer] $gid,
  String $comment,
) {
```

```
  user { $title:
    uid     => $uid,
    gid     => $uid,
    comment => $comment,
  }
}

# modules/users/manifests/init.pp
class users( Array[Hash] $userlist = [] ) {
  userlist.each do |$user| {
    users::create { $user['name']:
      uid     => $user['uid'],
      comment => $user['comment'],
    }
  }
}
```

The `create` defined type will be called once for every user in the array provided to the users module. Unlike a class, the defined type sees fresh input parameters each time it is called.

Calling Other Modules

In Building Subclasses we split up the module into separate subclasses for the Puppet agent and Puppet server. A complication of this split is that both the Puppet agent and Puppet server read the same configuration file *puppet.conf*. Both classes would modify this file, and both restart their services if the configuration changes.

Let's review two different ways to deal with this complication. Both solutions have classes depend on another module to handle configuration changes. Each of them present different ways to deal with the complications of module dependencies, thus we are covering both solutions to demonstrate different tactics.

Sourcing a Common Dependency

One way to solve this problem would be to create a third subclass named `config`. This module would contain a template for populating the configuration file with settings for both the agent and server. In this scenario, each of the classes could include the config class. This would work like the following:

```
# manifests/_config.pp
class puppet::_config(
  $agent = {},  # Agent params empty if not available in Hiera
  $server = {}, # Server params empty if not available in Hiera
) {
  file { 'puppet.conf':
    ensure  => ensure,
    path    => '/etc/puppetlabs/puppet/puppet.conf',
    owner   => 'root',
```

```
    group   => 'wheel',
    mode    => '0644',
    content => epp(
     'puppet:///puppet/puppet.conf.epp',        # template file
     { 'agent' => $agent, 'server' => $server } # hash of config params
    ),
  }
}
```

This example shows a common practice of naming classes which are used internally by a module with a leading underscore.

Best Practice: Name classes and types which should not be called directly by other modules with a leading underscore.

You may notice that the file resource doesn't `require` the agent or server packages, nor `notify` the Puppet agent or Puppet server services. This is because a Puppet agent and server are separate classes which might not be declared for a given node[2]. Those resources might not exist in the catalog.

Now let's modify the agent class to make use of this dependency.

```
# manifests/agent.pp
class puppet::agent(
  $status  = 'running',
  $enabled,
) {
  # Include the class which defines the config
  include puppet::_config

  # Install the Puppet agent
  package { 'puppet-agent':
    version => $version,
    before  => File['puppet.conf'],
    notify  => Service['puppet'],
  }

  # Manage the Puppet service
  service { 'puppet':
    ensure    => $status,
    enable    => $enabled,
```

2 The astute reader might point out that Puppet couldn't possibly configure the Puppet server if the Puppet agent isn't installed--a unique situation for only a Puppet module. This concept would be valid for any other module which handles both client and server.

```
    subscribe => [ Package['puppet-agent'], File['puppet.conf'] ],
  }
```

This example above uses `before` and `subscribe` to order the resources which must happen before or after the configuration is written out.

> You may think that `require` would be more appropriate than `include` for the `_config` class. However, require defines a complete dependency where every resource in the class depends on the other class. As we want the packages to be installed before the configuration file is modified, this would introduce a circular dependency. It is best to include the class and use per-resource dependencies.

If you imagine a server class defined the same way, this means that each one utilizes the same configuration class. As you might remember from the preceding section, each class is a singleton: the configuration class will only be called once, even though it is included by both classes. If the `puppet::server` class is defined with the same dependencies as the `puppet::agent` class, the `before` and `subscribe` attributes shown will ensure that implementation will happen in this order on a node which utilizes either or both classes:

1.
   - `puppet::agent`: The puppet agent package would be installed.
   - `puppet::server`: The puppet server package would be installed.
2. `puppet::_config`:The puppet configuration file would be written out.
3.
   - `puppet::agent`: The puppet agent service would be started.
   - `puppet::server`: The puppet server service would be started.

Using a Different Module

The previous example showed a way to solve a problem within a single Puppet module, where you control each of the classes which need to manage a common dependency. Sometimes there will be a common dependency shared across Puppet modules maintained by different groups, or perhaps even sometimes entirely outside of Puppet.

The use of templates requires the ability to manage the entire file. Even when using modules which can build a file from multiple parts, such the PuppetLabs Concat module, the entirety of the file must be defined within the Puppet catalog. The following example utilizes a module which can make individual line or section changes to a file without any knowledge of the remainder of the file.

```
# manifests/agent.pp
class puppet::agent(
  $version = 'latest',
  $status  = 'running',
  $enabled,
  $config  = {},   # Agent params empty if not available in Hiera
) {
  include puppetlabs::inifile

  # Fail if $config doesn't contain a hash
  validate_hash( $config )

  # Write each agent configuration option to the puppet.conf file
  $config.each |$setting,$value| {
    ini_setting { "agent $setting":
      ensure  => present,
      path    => '/etc/puppetlabs/puppet/puppet.conf',
      section => 'agent',
      setting => $setting,
      value   => $value,
      require => Package['puppet-agent'],
      notify  => Service['puppet'],
    }
  }

  package and service resources defined here
}
```

This example shows a way to define each configuration setting as a unique resource within the class. As each setting is a unique resource, the Package and Service won't use `before` or `subscribe` attributes as the config settings list can change. Instead the setting resources utiilze `require` and `notify` attributes to insert themselves between the package and service resources.

This shorter, simpler definition uses a third party module to update the Puppet configuration file in a non-exclusive manner. In my opinion this is significantly more flexible than the common dependency template module shown in the previous example.

Ordering Dependencies

As discussed in the section about variable scope, Classes and defined type instances contain the variables and resources they declare. An instance of a Class (singular) or defined type (multiple) becomes a container for the variables and resources defined within in.

This affects how the ordering metaparameters are applied. If a resource defines a dependency with a Class or Type, it will form the same relationship with every

resource inside the container. For example, say that we want the Puppet service to be started after rsyslog is already up and running. As you might imagine, the rsyslog module has a similar set of resources as our puppet::client module does:

```
class rsyslog {
  package { ... }
  file { ... }
  service { ... }
}
```

Rather than setting a dependencing on each one of these resources, we can set a dependency on the entire Class.

```
  # Manage the Puppet service
  service { 'puppet':
    ensure    => $status,
    enable    => $enabled,
    subscribe => Package['puppet-agent'],
    after      => Class['rsyslog'],
  }
```

With this definition, the Puppet service would not start until every resource in the `rsyslog` class has been processed.

Best Practice: Define dependencies on entire classes whenever possible. When you define dependencies on specific resources, a refactoring of the class could cause a compilation error for you.

It is possible to set a requirement on a specific instance of a defined type. To borrow the example of our user defined type

Containing Classes

In most situations, each class declaration stands independent. While a class can `include` another class, the class is defined at an equal level as the calling class--they are both instances of the `Class` type. Ordering metaparameters are used to control which classes are processed in which order.

As classes are peers, no class contains any other class. In almost every case, this is exactly how you want class declaration to work. This allows freedom for any class to set dependencies and ordering against any other class.

However, there is also a balance where one class should not be tightly tied to the internals of another class. It can be useful to allow other classes to declare ordering

metaparameters that refer to the parent class, yet ensure that any necessary subclasses are processed at the same time.

For example, our top-level puppet class contains only variables, and does not process any resources. A module which set a dependency like this would not achieve what they intended: `after => Class['puppet']`

Rather than require the module to set dependencies on each subclass of the Puppet module, we can define that each of the subclasses is contained within the main class using the `contain` function.

```
class puppet(
  # common variables for all Puppet classes
  String $version  = 'latest',
  String $loglevel = 'warning',
) {
  # Ensure that ordering includes subclasses
  contain 'puppet::agent'
  contain 'puppet::server'
}
```

With this definition, any class which declares an ordering metaparameter that references the `puppet` class need not be aware of the subclasses it uses.

Documenting the Module

In this section we're going to talk about how to document your manifests well. Good documentation ensures that others can use your module, or that you can recall what you were thinking when you come back to refactor your manifest a year later. Trust me, this happens.

Puppet 4 is deprecating RDoc syntax documentation in favor of using Markdown for all documentation. As this migration is still ongoing, we'll cover how to write documentation in both formats.

Learning Markdown

Puppet is moving away from RDoc format to the widely used Markdown format. Markdown is much easier to learn than RDoc, and is utilized today by the Puppet Forge, GitHub, Sourceforge, Stack Exchange, and many other code repositories and forums.

While you should absolutely read the Markdown documentation, it is entirely possible to build a valid and working README document with just the following eight simple rules:

1. Paragraphs should be typed as-is with no special formatting.

2. Code blocks should be indented four spaces.
3. Headers start with one # sign for each level. `#heading1 ##heading2`
4. Bullet lists start with a leading asterix, dash, or plus sign.
5. Number lists start with a leading number and period.
6. Use spaces to indent for list and code block hierarchy.
7. Surround words or phrases with single astericks for `*italic text*`.
8. Surround words or phrases with double astericks for `*bold text*`.

These eight rules provide more than enough syntax to create valid README documents.

Markdown supports a lot more syntax than this. You can find complete documentation of the format at http://daringfireball.net/projects/markdown/syntax.

Updating README.md

An initial *README.md* template is generated by the `puppet module generate` command. You should go through each of these sections and replace the example content with details specific to your module.

You can find Puppet Lab's latest recommendations for style at https://docs.puppetlabs.com/puppet/latest/reference/modules_documentation.html. However, we will review some important guidelines below.

One thing I have found to be unclear is the proper way to indicate compatibility. There are several parts to compatibility: operating system support, puppet requirements, and dependencies.

Indicating Compatibility

Operating system compatibility informs the viewer which operating systems and versions your module is known to work on. This is defined as an array of hashes in the metadata. A match for any one hash indicates success.

Each hash contains two values:

operatingsystem
: This is the value of `$facts['os']['name']`

operatingsystemrelease
: An array of possible values, which are matched against either `$facts['os']['release']['major']` or `"$facts['os']['release']['major'].$facts['os']['release']['minor']"`

Here's an example of compatibility which supports recent Enterprise Linux and Debian/Ubuntu versions.

```
"operatingsystem_support": [
  {
    "operatingsystem":"RedHat",
    "operatingsystemrelease":[ "6", "7" ]
  },
  {
    "operatingsystem":"CentOS",
    "operatingsystemrelease":[ "6", "7" ]
  },
  {
    "operatingsystem":"Amazon",
    "operatingsystemrelease":[ "2015.03", "2014.09", "2014.03" ]
  },
  {
    "operatingsystem":"OracleLinux",
    "operatingsystemrelease":[ "6", "7" ]
  },
  {
    "operatingsystem":"Scientific",
    "operatingsystemrelease":[ "6", "7" ]
  },
  {
    "operatingsystem":"Debian",
    "operatingsystemrelease":[ "6", "7" ]
  },
  {
    "operatingsystem": "Ubuntu",
    "operatingsystemrelease": [ "15.04", "14.10", "14.04", "13.10", "13.04" ]
  }
],
```

Defining Requirements

Requirements defines Puppet component versions. This is again an array of hashes. A match for any one hash indicates success.

name
: This is the value of the production. At this time I think only `puppet` and `pe` (puppet Enterprise) are supported.

version_requirement
: This is an expression which can utilize multiple <= and <= operators. For a module which only works in Puppet 4 you might use the example below, whereas a module which only supports Puppet 3 might use `>= 3.7.0 < 4.0.0`

```
"requirements": [
  { "name": "pe", "version_requirement": ">= 4.0.0" },
  { "name": "puppet", "version_requirement": ">= 4.0.0" }
],
```

I wish there was a way to suggest that Puppet 3 versions which utilized the *future* parser (the prototype for Puppet 4's parsing engine) were compatible, but I've found no way to express that.

Listing Dependencies

Dependencies lists out other Puppet modules which are necessary for this module to function. This is also defined as an array of hashes. However, unlike the previous two arrays, every one of the dependencies must be met.

name
: This is the name of the puppet module as shown on the forge, however with the prefix and the module name separated by a `/` slash.

version_requirement
: This is an expression which can utilize multiple `<=` and `<=` operators to indicate a valid range of matching versions.

```
"dependencies": [
  { "name": "puppetlabs/stdlib", "version_requirement": ">= 3.2.0" }
]
```

Creating CHANGELOG.md

This file isn't generated for you by default, but you should create this file and update it with every version change in your module. For each version change, include something like this:

```
##YYYY-MM-DD - Release X.Y.Z
###Summary

This release ...

####Features
- Added new...
- Revised...

####Bugfixes
- Fixed bug where...
```

Documenting the Classes and Types

Each class and defined type in your module should be documented.

YARD Markdown

Puppet 4 has moved away from RDoc in favor of Markdown format documentation for consistency. You can use Markdown format even if your module is used by Puppet 3 users, as Markdown is easy to read and Puppet 3 users can install the `puppet-strings` module to generate HTML and PDF documentation.

`puppet doc` no longer generates module documentation in Puppet 4. Module documentation is generated by `puppet strings`, which is made available by installing the `puppetlabs-strings` module.

First, we'll install the Puppet Labs strings module:

```
$ puppet module install puppetlabs-strings
Notice: Preparing to install into /home/vagrant/.puppetlabs/etc/code/modules ...
Notice: Created target directory /home/vagrant/.puppetlabs/etc/code/modules
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/home/vagrant/.puppetlabs/etc/code/modules
└── puppetlabs-strings (v0.2.0)

$ gem install yard --no-ri --no-rdoc
Fetching: yard-0.8.7.6.gem (100%)
Successfully installed yard-0.8.7.6
1 gem installed
```

Puppet Strings can process both Markdown and RDOC documentation like so:

```
$ cd /etc/puppetlabs/code/environments/testing/modules/puppet
$ puppet strings
iles:           1
Modules:         1 (    0 undocumented)
Classes:         3 (    0 undocumented)
Constants:       0 (    0 undocumented)
Methods:         0 (    0 undocumented)
Puppet Classes:      3 (    0 undocumented)
Puppet Types:      1 (    0 undocumented)
 100.00% documented
true
```

Puppet Strings generates HTML documentation in the *doc/* folder within the module.

Parameters. Parameters are documented using `@param` meta tag. Follow this tag with the name of the parameter and a description of it's use. The following lines should document the default and expected values.

Here is an example for documenting the `puppet::client` class we built earlier in the book.

```
# @param status Whether Puppet client should run as a daemon
#    values: running (default), stopped
# @param enabled Whether Puppet client should start at boot
#    values: true, false (value required)

class puppet::agent(
  $status  = 'running',
  $enabled,
) {
```

Examples. Examples are documented using `@examples` meta tag. Follow this tag with the name of the parameter and a description of its use. The following lines should document how to use the module, the required values, and the most likely use case.

Here is an example for documenting how to use the `puppet::client` class we built earlier in the book.

```
# @examples Hiera data
#   classes:
#     - puppet::client
#   puppet::client::status = 'running'
#   puppet::client::enabled = true
#
```

Evolving Effort. The migration to Markdown format is an evolving effort which will likely iterate a few times during the development of this book, and continue after it has finished.

You can find the latest updates for recommended style at puppetlabs-strings module documentation.

Puppet Strings utilizes the YARD gem for formatting, so it can be useful to refer to the YARD Documentation.

Authors and Copyright. The Authors section of the documentation is self explanatory. List your name and the names of your co-authors. Include an e-mail address they can contact you at with questions. If intend to publish this module on the Puppet Forge and don't feel comfortable giving out your e-mail, you could list the address of your issue tracking system as a contact method.

Here's an example:

```
# ###Authors
# - Jo Rhett, bug reports accepted at http://github.com/jorhett/puppet-module/issues
#
```

For the copyright section of the module list the copyright. If this is an internal module that you won't be publishing online, you can use ***Company Name*, All Rights Reserved.** If you are publishing on the Puppet Forge, you should use a license which allows others to use the module, such as one of the following:

- Apache 2.0 License http://www.apache.org/...
- LGPL License http://gnu.org/...
- BSD License http://...

> If you have done this work for an organization which has employed or contracted with you, they likely have ownership of your work. It can be even more complicated if you work for an organization who is providing services to another organization.
>
> Anytime someone else's rights are involved, you should get legal advice before you publish the work in any form. There are lots of ways to arrange for the appropriate permission, but as the author I am going to duck this topic because **I Am Not A Lawyer**. You and I are both competent technical professionals. We should listen to the people who are competent in a completely different profession.

```
# ###Copyright
# Copyright Acme Products, 2015
# All Rights Reserved
#
```

RDoc

When you generated the Puppet module earlier in this book, the *init.pp* file already contained some sample documentation. These samples are written in RDoc, the documentation format required by the Puppet Style Guide, and utilized by `puppet doc` and `puppet-lint`.

Although the Puppet Strings Markdown syntax will eventually replace RDoc, the new style is still being actively developed and the new standards are just now coming into shape. If you are comfortable using RDoc format, or if you need to update an existing module, we've included instructions here. This is also widely utilized in existing Puppet modules, so expect to see and provide this format for a long time.

If you are new to Puppet and starting with Puppet 4, you can safely skip this section.

Parameters. Parameters are documented using `[*param*]` on a line by itself. Follow this line with a freeform description of the parameter and its default and expected values.

Here is an example for documenting the `puppet::client` class we built earlier in the book.

```
# === Parameters
#
# [*status*]
#   Whether Puppet client should run as a daemon
#    values: running (default), stopped
# [*enabled*]
#   Whether Puppet client should start at boot
#    values: true, false (value required)
#
class puppet::agent(
  $status  = 'running',
  $enabled,
) {
```

Variables. In the Variables section of the documentation, list any variables used by this class which are not provided by the class parameters. This can include:

- Direct access of variables from other modules and classes.
- Direct hiera lookups of data not passed as parameters.

Variables are documented using `[*variable*]` on a line by itself. Follow this line with a freeform description of the variables and its default and expected values.

Here is an example for documenting the `puppet::client` class we built earlier in the book.

```
# === Variables
#
# [*puppet::version*]
#   This class uses the common $version variable shared by all Puppet subclasses.
#
class puppet::agent(...) {
  include '::puppet'

  package { 'puppet-agent':
    version => $::puppet::version,
  }
```

Examples. Examples are documented using `@examples` meta tag. Follow this tag with the name of the parameter and a description of its use. The following lines should document how to use the module, the required values, and the most likely use case.

Here is an example for documenting how to use the `puppet::client` class we built earlier in the book.

```
# === Examples
#
```

```
# Hiera data:
#   classes:
#     - puppet::client
#   puppet::client::status = 'running'
#   puppet::client::enabled = true
#
```

Authors and Copyright. The Authors section of the documentation is self explanatory. List your name and the names of your co-authors. Include an e-mail address they can contact you at with questions. If intend to publish this module on the Puppet Forge and don't feel comfortable giving out your e-mail, you could list the address of your issue tracking system as a contact method.

Here's an example:

```
# === Authors
#
# Jo Rhett, bug reports accepted at http://github.com/jorhett/puppet-module/issues
#
```

For the copyright section of the module list the copyright. If this is an internal module that you won't be publishing online, you can use ***Company Name*, All Rights Reserved.** If you are publishing on the Puppet Forge, you should use a license which allows others to use the module, such as one of the following:

- Apache 2.0 License http://www.apache.org/...
- LGPL License http://gnu.org/...
- BSD License http://...

> See the essential comments about code ownership in the Puppet 4 Docs section.

```
# === Copyright
#
# Copyright Acme Products 2015
# All Rights Reserved
#
```

Peeking Beneath the Hood

In this section we're going to talk about peeking beneath the hood; looking at variables and resources in other classes.

Best Practice: Don't use anything described here for anything other than debugging purposes.

Class boundaries are not enforced. You can access both variables and resources in other classes. Here's an example:

```
class other_class( String $idea ) {
  $sentence = "an idea ${idea} wrapped in other_module's namespace"
  notify { 'announcement': message => "I have ${sentence}" }
}

class my_class {
  # I can see the other class parameter
  notice( "The idea was: ${Class['other_class']['idea']}" )

  # I can see the other class variables
  notice( "The sentence was: ${::other_class::sentence}" )

  # I can see parameters of resources in another class
  notice( "The entire message was: ${Notify['announcement']['message']}" )
}
```

Given an idea of *games!* you'd see output like this:

```
Notice: Scope(Class[My_class]): The idea was: games!
Notice: Scope(Class[My_class]): The sentence was: an idea 'games!' wrapped in other_module's names
Notice: Scope(Class[My_class]): The entire message was: I have an idea 'games!' wrapped in other_m
```

In Puppet4 visibility is limited:

- Puppet Runtime defines are visible to everything - everything in an environment is visible to everything in the same environment - modules without dependencies see all other modules - modules with dependency see the module it depends on

Best Practices for Module Design

Let's review some of the best practices for module development we covered in this section:

- Declare a class with the module name in *manifests/init.pp*.
- Place every subclass and defined type in a separate manifest file with the name of the class or type.
- Update the README and CHANGELOG Markdown documents for each new version.
- Document every manifest using Markdown (or RDoc) markup within the file.
- Validate each and every parameter for expected values in the manifest.
- Assign default values for parameters that reflect most likely case.

- Avoid using top-level or nodes variables.
- Reference variables from other classes with empty namespace prefix `$::other module::variable`
- Declare parameters with explicit types whenever possible for data validation.
- Create a test for each manifest in the *tests/* directory.
- Tests should validate the most common case and default parameter values for every class and defined type.

You can find more detailed guidelines in the Puppet Labs Style Guide.

Modules Review

Modules provide an independent namespace for reusable blocks of code which configure or maintain something. A module provides new resource types which can be independently used by others.

In this chapter we have covered how to configure a module to:

- Provide reusable Puppet code others can utilize with their own data
- Accept parameters from Hiera and ENCs
- Synchronize files and directories to the managed nodes
- Customize files from templates and node data
- Utilize other modules to provide dependencies
- Share new types and subclasses with discrete functionality
- Provide documentation for users of the module

This has covered the required pieces of modules, and the most common use cases for them. In the next section we're going to cover how to create system and unit tests for a module.

CHAPTER 14

Testing Modules

Sad to say, but not many modules include good tests. Good tests help you avoid embarrassing bugs from going out. Good tests save you a lot of time, avoiding the exhaustive debugging of an issue in production which turns out to be a wrong type used at the wrong point.

This chapter will teach you how to add good tests to your modules. When I got started with them I struggled a lot due to a lack of good examples. In this section I'm going to provide good examples of each type of test you should be doing. It's my intention that you'd be able to use the examples provided here like tinker toys, and build a good set of tests for your modules without much effort.

This chapter won't provide exhaustive documentation of `rspec` or `beaker`, the testing tools of choice for the Puppet ecosystem. However, you should be able to build a good foundation of tests from what we cover in this chapter.

Let's get started by setting up your testing tools.

Installing Dependencies

The first time you set up to do testing, you'll need to install some software used for testing.

Installing Ruby

You can use the version of Ruby which comes with your operating system. If you are using the Vagrant testing setup documented in this book, it is easy to install Ruby into the system packages.

```
[vagrant@client ~]$ sudo yum install -y ruby rubygems rake
```

If you are using an older operating system which doesn't have a modern Ruby 2.0 or higher available, you can install the software dependencies into the Ruby which came with Puppet's All-in-One (AIO) installer. This has a complication in that binaries are installed outside of your path. For convenience when using the ruby provided with Puppet, run the following commands to set up symbolic links for simplicity:

```
[vagrant@client ~]$ sudo alternatives --install /usr/bin/ruby ruby /opt/puppetlabs/puppet/bin/ruby
[vagrant@client ~]$ sudo alternatives --install /usr/bin/gem gem /opt/puppetlabs/puppet/bin/gem 5
[vagrant@client ~]$ sudo alternatives --install /usr/bin/rake rake /opt/puppetlabs/puppet/bin/rake
```

While I have found it handy to install the necessary Ruby gems with the modern Ruby provided by Puppet, this is not a supported configuration. I would use this only on test and development nodes, and never on a production systems.

Installing Gem Bundler

The bundler gem will help you install any and all necessary dependencies.

```
$ sudo gem install bundler
Fetching: bundler-1.10.3.gem (100%)
Successfully installed bundler-1.10.3
1 gem installed
```

If you are using gem from the Ruby installed by puppet, you may want to add the bundle commands to your path as well. For convenience, run the following commands to set up symbolic links:

```
$ sudo alternatives --install /usr/bin/bundle bundle /opt/puppetlabs/puppet/bin/bundle 5
$ sudo alternatives --install /usr/bin/bundler bundler /opt/puppetlabs/puppet/bin/bundler 5
```

Installing Spec Helper

If you haven't done this already, you'll need to install the *puppetlabs_spec_helper* and other dependency gems. The best way to do this is to run the `bundler` gem we just installed from within your module directory. It reads the dependency versions from the *Gemfile* and will ensure the correct versions of each gem is installed.

Bundler will pull in `rspec`, `rspec-puppet`, and all of their dependencies as well. These are testing and template creation tools which help simplify test creation.

```
[vagrant@client puppet]$ bundler install
Fetching gem metadata from https://rubygems.org/.........
Fetching version metadata from https://rubygems.org/..
Resolving dependencies...
```

```
Installing rake 10.4.2
Installing CFPropertyList 2.2.8
Using diff-lcs 1.2.5
Installing facter 2.4.4
Installing json_pure 1.8.2
Installing hiera 2.0.0
Using metaclass 0.0.4
Using mocha 1.1.0
Installing puppet 4.1.0
Using puppet-lint 1.1.0
Using puppet-syntax 2.0.0
Using rspec-support 3.2.2
Using rspec-core 3.2.3
Using rspec-expectations 3.2.1
Using rspec-mocks 3.2.1
Using rspec 3.2.0
Installing rspec-puppet 2.2.0
Using puppetlabs_spec_helper 0.10.3
Using bundler 1.10.3
Bundle complete! 4 Gemfile dependencies, 19 gems now installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

Don't use sudo when running bundler.

Preparing Your Module

The next step is to setup your module for testing. We'll have to modify a few files to use the best tools for this.

Defining Fixtures

Create a *.fixtures.yml* file which defines the testing fixtures (dependencies) and where to acquire them for testing purposes. The information in this file should duplicate the dependencies in *metadata.json*.

The top of the file should always be the same. This tells the testing frame to copy the current module from the module directory.

```
fixtures:
  symlinks:
    puppet: "#{source_dir}"
```

Then define each dependency for your module and the minimum version you support. You can list their name on the Puppet Forge or their Github URL. The following two examples will have similar effects. From the Forge:

```
forge_modules:
  stdlib:
    repo: "puppetlabs/stdlib"
    ref: 4.5.1
```

From GitHub:

```
repositories:
  stdlib:
    repo: "git://github.com/puppetlabs/puppetlabs-stdlib"
    ref: "4.5.1"
```

If you are testing development of multiple modules, you may want to use symlinks to the source tree for each. Assuming the dependency is in the same directory structure:

```
symlinks:
  some_dependency: "#{source_dir}/../some_dependency"
```

Test that dependency setup worked properly like so:

```
$ rake spec
(in /etc/puppetlabs/code/environments/testing/modules/puppet)
Notice: Preparing to install into /etc/puppetlabs/code/environments/testing/modules/puppet/spec/fi
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/code/environments/testing/modules/puppet/spec/fixtures/modules
└── puppetlabs-stdlib (v3.2.1)
/usr/bin/ruby -I/usr/lib/ruby/gems/1.8/gems/rspec-support-3.2.2/lib:/usr/lib/ruby/gems/1.8/gems/rs
No examples found.

Finished in 0.00027 seconds (files took 0.04311 seconds to load)
0 examples, 0 failures
```

This shows that all fixtures (dependencies) were installed, but no examples (tests) were available. Let's start building one now.

Defining Tests

Now let's build some tests for the module. We know, few people think that building tests is fun work--but it is important work that will save you time and effort down the road.

Following are guidelines for building useful tests. Let's go over them now.

- Test every input parameter.
- Test every file, package, and service name.
- Test every variation in implementation your module is designed to handle.
- Test for implicit choices based around operating system or other environmental tests.

- Test for invalid input as well as valid input.

Let's look at some examples testing each one of these situations.

Defining Main Class

Within your module directory, change into the *spec/classes/* directory. Inside this directory, create a file named *modulename_spec.rb*.

```
[vagrant@client puppet]$ cd spec/classes
[vagrant@client classes]$ $EDITOR puppet_spec.rb
```

For our initial test, we will simply build one test that the module compiles successfully with the default options.

```
require 'spec_helper'

describe 'puppet', :type => 'class' do

  context 'with defaults for all parameters' do
    it do
      should contain_class('puppet')
      should contain_class('puppet::params')
    end

    it do
      should compile.with_all_deps
    end
  end
```

Let's go ahead and run the testing suite against this very basic test.

```
[vagrant@client puppet]$ rake spec
(in /etc/puppetlabs/code/modules/puppet)
ruby -I/opt/puppetlabs/puppet/lib/ruby/gems/2.1.0/gems/rspec-support-3.2.2/lib:/opt/puppetlabs/pup
..

Finished in 10.22 seconds (files took 0.56629 seconds to load)
2 examples, 0 failures
```

At this time you may see the following error:

```
Failures:

  1) puppet with defaults for all parameters should contain Class[puppet]
```

This is due to an older version of the puppet gem being loaded. You'll need at least version 3.7. Any of the following commands will solve your problem. If you are testing modules for maximum compatibility, run all of these commands. Notice the special comparison operator ~> which will give you the latest available version for each minor version of Puppet.

```
$ gem install puppet --no-ri --no-rdoc
Fetching: puppet-4.1.0.gem (100%)
Successfully installed puppet-4.1.0
1 gem installed

$ gem install puppet --version '~> 3.8.0' --no-ri --no-rdoc
Fetching: puppet-3.8.1.gem (100%)
Successfully installed puppet-3.8.1
1 gem installed

$ gem install puppet --version '~> 3.7.0' --no-ri --no-rdoc
Fetching: puppet-3.7.5.gem (100%)
Successfully installed puppet-3.7.5
1 gem installed
```

Try running your test again, and you should see that it succeeds.

Now that our basic test passed, let's go on to start checking the input parameters.

Passing Valid Parameters

How about if we expand the tests to include every possible input value? Rather than repeating each test with a different value, we build Ruby loops to iteratively build each of the tests from an array of values.

```
['1','0'].each do |repo_enabled|
    ['emerg','crit','alert','err','warning','notice','info','debug','verbose'].each do |loglevel|
      context "with #{repo_enabled} for repo_enabled, #{loglevel} for loglevel" do
        let :params do
          {
            :repo_enabled => repo_enabled,
            :loglevel     => loglevel,
          }
        end

      it do
        should contain_package('puppet-agent').with({
          'version' => '1.1.0-1'
        })
      end
    end
  end
 end
```

Woah, look at that. We added 17 lines of code and yet it's performing 36 more tests now.

```
[vagrant@client puppet]$ rake spec
(in /etc/puppetlabs/code/modules/puppet)
ruby -I/opt/puppetlabs/puppet/lib/ruby/gems/2.1.0/gems/rspec-support-3.2.2/lib:/opt/puppetlabs/pup
```

```
....................................

Finished in 10.53 seconds (files took 0.56829 seconds to load)
38 examples, 0 failures
```

Failing Invalid Parameters

Now let's test to ensure some incorrect values fail. So here we define two tests that are intended to fail.

```
  context 'with invalid loglevel' do
    let :params do
      {
        :loglevel => 'annoying'
      }
    end

    it do
      expect { should compile.with_all_deps }
    end
  end

  context 'with invalid repo_enabled' do
    let :params do
      {
        :repo_enabled => 'EPEL'
      }
    end

    it do
      expect { should compile.with_all_deps }
    end
  end
```

Now we should run the tests to see what error messages are kicked back.

```
[vagrant@client puppet]$ rake spec
Failures:

  1) puppet4 with invalid loglevel should compile into a catalogue without dependency cycles
     Failure/Error: should compile.with_all_deps
       error during compilation: Parameter loglevel failed on Class[Puppet]: Invalid value "annoyi
     # ./spec/classes/puppet_spec.rb:52:in `block (3 levels) in <top (required)>'

  2) puppet4 with invalid repo_enabled should compile into a catalogue without dependency cycles
     Failure/Error: should compile.with_all_deps
       error during compilation: Expected parameter 'repo_enabled' of 'Class[Puppet]' to have type
     # ./spec/classes/puppet_spec.rb:65:in `block (3 levels) in <top (required)>'

Finished in 10.81 seconds (files took 0.57989 seconds to load)
40 examples, 2 failures
```

Now let's change the expect lines for the errors we are expecting:

```
expect { should raise_error(Puppet::Error,/Invalid value "annoying". Valid values are/) }
```

...and the following for the repo_enabled test:

```
expect { should raise_error(Puppet::Error,/Expected parameter 'repo_enabled' .* to have type
```

Now when you run the tests, you will see the tests were successful because the bad tests failed.

Adding an Agent Class

Within the *spec/classes/* directory, create a file named *agent_spec.rb*. This is an exercise for you. Build the agent class, testing every valid and invalid input just like we did for the puppet class.

For this we simply want to test that the package, config file, and service resources are all defined.

```
require 'spec_helper'

describe 'puppet::agent', :type => 'class' do

  context 'with defaults for all parameters' do
    it do
        should contain_package('puppet-agent').with({ 'version' => 'latest' })
        should contain_file('puppet.conf').with({ 'ensure' => 'file' })
        should contain_service('puppet').with({ 'ensure' => 'running', 'enabled => true })
    end

    it do
        should compile.with_all_deps
    end
  end
end
```

We have demonstrated the tests. Now build out new tests for valid and invalid input.

Using Hiera Input

Within your module directory, change into the *spec/fixtures/* directory. Inside this directory, create a subdirectory named *hiera*, containing a valid *hiera.yaml* file for testing.

```
[vagrant@client puppet]$ cd spec/fixtures
[vagrant@client puppet]$ mkdir hiera
[vagrant@client classes]$ $EDITOR hiera/hiera.yaml
```

You can change anything you want that is valid for Hiera in this configuration file, except for the `datadir`, which should reside within the fixtures path. Unless you desire a specific change, the following file could be used unchanged in every module:

```
# spec/fixtures/hiera/hiera.yaml
---
:backends:
  - yaml
:yaml:
  :datadir: /etc/puppetlabs/code/hieradata
:hierarchy:
  - defaults
  - "%{facts::osfamily}"
  - global
```

Now, add the following lines to a test context within one of the class spec files:

```
  let(:hiera_config) { 'spec/fixtures/hiera/hiera.yaml' }
  hiera = Hiera.new( :config => 'spec/fixtures/hiera/hiera.yaml' )
```

Now create your Hiera input files. The only necessary file is *spec/fixtures/hiera/global.yaml*. The others can be added only when you want to test things.

```
---
puppet::loglevel      : 'notice'
puppet::repo_enabled  : '1'
puppet::agent::status : 'running'
puppet::agent::enabled: true
```

This hiera data can be used when configuring the tests:

```
  let :params do
    {
      :repo_enabled => hiera.lookup('puppet::repo_enabled',nil,nil),
      :loglevel     => hiera.lookup('puppet::loglevel',nil,nil),
    }
  end
```

This configuration allows you to easily test the more common mode of using hiera-input parameters for your modules.

Defining Parent Class Parameters

In some situations your module will depend upon a class that requires some parameters to be provided. You cannot set parameters or use Hiera for that class, since it is out of scope for the current class and test file.

The workaround is to use a `pre_condition` block to call the parent class in resource-style format. Pass the necessary parameters for testing as parameters for the resource declaration, and this module instance will be created before your module is tested.

Here is an example from my *mcollective* module, which had to solve exactly this problem.

```
describe 'mcollective::client' do
  let(:pre_condition) do
    'class { "mcollective":
      hosts           => ["middleware.example.net"],
      client_password => "fakeTestingClientPassword",
      server_password => "fakeTestingServerPassword",
      psk_key         => "fakeTestingPreSharedKey",
    }'
  end

...tests for the mcollective::client class...
```

Improve Testing with Custom Skeletons

There are a number of puppet skeletons that include frameworks for enhanced testing above and beyond what we've covered. You may to tune the module skeleton you use to include testing frameworks and datasets consistent with your release process. Place the revised skeleton in the *~/.puppetlabs/opt/puppet/cache/puppet-module/skeleton* directory, or specify it on the generate command line with `--module_skeleton_dir=`*`path/to/skeleton`*.

Following are some skeletons I have found useful at one time or another:

garethr/puppet-module-skeleton

> This skeleton is very opinionated. It's going to assume you're going to start out with tests (both unit and system), that you care about the puppet style guide, test using Travis, keep track of releases and structure your modules according to strong conventions.
>
> —https://github.com/garethr/puppet-module-skeleton

This is a popular and widely used skeleton.

jimdo/puppet-skeleton

> The module comes with everything you need to develop infrastructure code with Puppet and feel confident about it.
>
> —https://github.com/jimdo/puppet-skeleton

This skeleton includes helpers to spin up Vagrant instances and run tests on them.

ghoneycutt/puppet-module-skeleton

At the time this book was written, Garret didn't have a README for this skeleton at all -- however Garret is an active and high-quality contributor to the Puppet community.

gds-operations/puppet-skeleton

> This is a skeleton project for Web Operations teams using Puppet. It ties together a suite of sensible defaults, best current practices, and re-usable code. The intentions of which are two-fold: * New projects can get started and bootstrapped faster without needing to collate or re-writing this material themselves. * The standardisation and modularisation of these materials makes it easier for ongoing improvements to be shared, in both directions, between different teams.
>
> —https://github.com/gds-operations/puppet-skeleton

wavesoftware/puppet-os-skeleton

> A complete working solution with:
>
> - Puppet master and agent nodes on Puppet Open Source
> - Spotify Puppet Explorer and PuppetDB
> - Hiera configuration
> - Dynamic GIT environments by r10k
> - External puppet modules installation and maintenance by r10k
> - Landrush local DNS
>
> Coupe of bootstrap puppet classes:
>
> - common::filebucket - use of filebucket on all files
> - common::packages - central packages installation from hiera
> - common::prompt - a Bash command prompt with support for Git and Mercurial
>
> —https://github.com/wavesoftware/puppet-os-skeleton

You can find many others by searching for "puppet skeleton" on GitHub. In particular, you can find skeletons specialized for the frameworks the application is built in, e.g. OpenStack, Rails, Django, etc.

Simplifying with Tools

There are a number of tools which make it easier to setup your tests, or perform additional tests, or plug in better with other testing frameworks. Let's examine some of these.

Puppet-Retrospect

Puppet Retrospect provides a tool which will create basic tests for each of your manifests.

> Retrospec makes it dead simple to get started with puppet unit testing. When you run retrospec, retrospec will scan your puppet manifests and write some

> very basic rspec-puppet test code. Thus this gem will retrofit your existing puppet module with everything needed to get going with puppet unit testing.
>
> —https://github.com/logicminds/puppet-retrospec

In the author's experience, this is not a final solution by itself but a way to generate a useful skeleton of tests. This is a good tool to run when you haven't started testing. I often use this tool when I'm going back to add tests to an existing module that doesn't have any. I then take what it generates and tweak the tests and add more comprehensive tests, as documented in the previous section.

Finding Documentation

You may have found a tricky problem not covered by the examples here. At this point it is best to refer to the original vendor documentation:

- RSpec: Behavior-Driven Development for Ruby
- RSpec Tests for Puppet extension
- Puppet Labs Spec Helper: Shared Spec Helpers for Puppetlabs Projects
- Beaker: Puppet Acceptance Testing Harness
- RSpec Best Practices

Much of this documentation is dated, but still valid. There are open bugs to provide Puppet 4.x-specific documentation, and I will update this section as soon as it is available.

Testing Modules Review

Each class and defined type should have tests defined for it. In this chapter we have covered how to test modules for:

- Simple compilation success with default values.
- Minimum and acceptable values passed in as parameters.
- Creation of the resources they were intended to manage.
- Invalid and unacceptable values.
- Providing data using Hiera fixtures.
- Preloading parent modules with required parameters to ensure module dependencies are valid.

This has covered the necessary tests which should be included in every module. In the next section we're going to cover how to create plugins to extend modules with less common functionality.

CHAPTER 15

Extending Modules with Plugins

In this chapter we are going to cover adding plugins to puppet modules. Plugins are used to provide new Facts, Functions, and module-specific data which can be used in the Puppet catalog.

Nothing in this chapter is required to build a working module, and there are thousands of modules which don't use plugins. You can safely skip this chapter and return back after you are comfortable building the common features of modules.

Many of the extensions in this chapter are written in Ruby. To build Ruby plugins, knowledge of the Ruby programming language is required. I recommend Learning Ruby as an excellent reference for this.

Adding Custom Facts

One of the most useful plugins a module can provide is custom Facts. These are facts not provided by Facter, but custom to your module. Plugin facts are synced down to the node in Master/agent environments, and available to the Puppet agent during the convergence process for each Puppet run thereafter.

Previous versions of Puppet could only supply string values for facts. In Puppet 4, custom facts can return any of the Scalar data types (e.g. String, Numeric, Boolean, etc) in addition to any of the Collection types (Array, Hash, Struct, etc).

There are two ways to provide custom facts: using Ruby functions, or through external data. Let's go over how to build new facts using both methods.

External Facts

Would you like to provide facts without writing Ruby? There are two ways to do this:

1. Write fact data out in YAML, JSON, or text format.
2. Provide a program or script to output fact names and values.

The program or script can be written in Bash, Python, Java, whatever. It's only necessary that it can be executed by the Puppet agent.

This is what the new External Facts are for. Let's go over how to use these.

Structured Data

You can place structured data files in the *facts.d/* directory of your module with data to assign to facts. Structured data files must be in a known formats, and must be named with the appropriate file extension for their format.At the time this book was written, the following formats were supported:

Type	Extension	Description
YAML	*.yaml*	Facts in YAML format
JSON	*.json*	Facts in JSON format
Text	*.txt*	Facts in key=value format.

We introduced YAML format back in Separating Data from Code. Following is a simplified YAML example that sets three facts.

```
# three_simple_facts.yaml
---
my_fact: myvalue
my_effort: easy
is_dyanamic: false
```

The text format uses a single *key=value* pair on each line of the file. This only works for String values, arrays and hashes are not supported. Here is the same example in text format:

```
# three_simple_facts.txt
my_fact=myvalue
my_effort=easy
is_dyanamic=false
```

Programs

You can place any executables program or script in the *facts.d/* directory of your module to create new facts.

The script or program must have the execute bit set for root, or the user that you are running puppet as. The script must output each fact as *key=value* on a line by itself. Following is a simplified example that sets three facts.

```
#!/bin/bash
echo "my_fact=myvalue"
echo "my_effort=easy"
echo "is_dyanamic=false"
```

Install this in the directory and test it.

```
$ $EDITOR facts.d/three_simple_facts.sh
$ chmod 0755 facts.d/three_simple_facts.sh
$ facts.d/three_simple_facts.sh
my_fact=myvalue
my_effort=easy
is_dyanamic=false
```

Windows. Executable facts on Windows work exactly the same, and require the same execute permissions and output format. However the program or script must be named with a known extension. At the time this book was written, the following extensions were supported:

.com, .exe
: binary executables to be executed directly

.psl PowerShell Scripts
: scripts to be run by the PowerShell interpreter

.cmd, .bat Command Shell Scripts
: ASCII or UTF8 batch scripts to be processed by `cmd.exe`

Following is the same example from the previous page rewritten as a PowerShell script:

```
Write-Host "my_fact=myvalue"
Write-Host "my_effort=easy"
Write-Host "is_dyanamic=false"
```

You should be able to save and execute this PowerShell script on the command line.

Debugging

Your new fact will not appear in the output of `facter`. To see the values of Puppet facts you'll need to use `puppet facts find` instead.

If your external fact is not appearing in Facter's output, running Facter in debug mode should give you a meaningful reason and tell you which file is causing the problem:

```
$ puppet facts find --debug
...
Debug: Loading facts from /etc/puppetlabs/code/modules/stdlib/lib/facter/facter_dot_d.rb
Debug: Loading facts from /etc/puppetlabs/modules/stdlib/lib/facter/pe_version.rb
Debug: Loading facts from /etc/puppetlabs/code/modules/stdlib/lib/facter/puppet_vardir.rb
Debug: Loading facts from /etc/puppetlabs/code/environment/testing/modules/puppet/facts.d/three_si
```

If the output from your custom fact wasn't in the proper format, you'll get errors like this:

```
Fact file /etc/puppetlabs/code/environment/testing/modules/puppet/facts.d/python_sample.py was par
```

In those situations, run the program by hand and examine the output. Here's the example I used to generate the complaint above:

```
$ /etc/puppetlabs/code/environment/testing/modules/puppet
$ facts.d/python_sample.py
fact1=this
fact2
fact3=that
```

The external facts per module functionality obsoletes and supercedes the less powerful `facter-dot-d` functionality provided by the *stdlib* module. If you were using facts installed in the node global *facter/facts.d* directory, move them out of there and into an appropriate *modules/modulename/facts.d* directory.

Custom (Ruby) Facts

To create new facts in Ruby for Facter to use, simply create the following directory in your module:

```
$ mkdir -p lib/facter
```

Any ruby programs in this directory will be synced down to client node at the beginnning of the Puppet run. Within this directory create a ruby program ending in *.rb*. The ruby program can contain any normal Ruby code, however the process of defining a custom fact is implemented by two calls:

1. A Ruby code block starting with `Facter.add('fact_name')`
2. Inside the Facter code block, a `setcode` code block which returns the fact's value.

For an easy example, let's assume that your hosts are grouped in clusters with the same name. Unique numbers added to each node to keep them distinct. This results in a common pattern with host names like:

- webserver01

- webserver02
- webserver03
- mailserver01
- mailserver02
- *...etc*

You'd like to define Hiera data based on the cluster name. Right away, one might think to use the `hostname` command to acquire the node name. Facter provides a helper function to execute shell commands: `Facter::Core::Execution.exec()`.

Keep in mind that this is Ruby code, and what you are passing to the function is a Ruby String. You should escape meta characters as required by Ruby, rather than the more permissive rules of Puppet.

The code to derive a node's cluster name from the `hostname` shell command could look like this:

```
# cluster_name.rb
Facter.add('cluster_name') do
  setcode do
    Facter::Core::Execution.exec("/bin/hostname -s | /usr/bin/sed -e 's/\d//g'")
  end
end
```

This new fact will be available as `$facts['cluster_name']` in your manifests and templates during the next Puppet run.

The value of the string should be a command to execute. Pipe `|` and redirection `>` operators work as you might expect, however you don't have full shell scripting available. Shell control structures like `if` or `for` do not work. Best practice is to run a single command to acquire a value, and then evaluate or manipulate the value using native Ruby code.

Here's an example using Ruby native libraries to acquire the hostname, and then manipulate the value to remove the domain and the numbers:

```
# cluster_name.rb
require 'socket'
Facter.add('cluster_name') do
  setcode do
    hostname = Socket.gethostname
    hostname.sub!(/\..*$/, '') # remove the first period and everything after it
    hostname.gsub(/[0-9]/, '') # remove every number and return revised name
  end
end
```

For even more optimization, you could also refer to the existing hostname fact and use that. You can acquire the value of an existing fact using `Facter.value('fact name')`. Here is a simpler example starting with the existing hostname fact.

```
# cluster_name.rb
Facter.add('cluster_name') do
  setcode do
    hostname = Facter.value(:hostname).sub(/\..*$/, '')
    hostname.gsub(/[0-9]/, '')
  end
end
```

If you are running Puppet with a Puppetmaster or Puppet Server (covered in the next part of this book), facts from modules are synced down to the agent node during the agent configuration phase. You can run `puppet facts` to see the facts that have been distributed via pluginsync.

Avoiding Delay

To limit problems with code which may run long or hang, use the `timeout` property of `Facter.add()` to define how many seconds the code should complete within. This causes the `setcode` block to halt if the timeout is exceeded. The puppet run will move on without an error, but also without a value for the fact. This is generally preferable to a hung Puppet client.

Returning to our example of calling a shell command, modify it as follows:

```
# cluster_name.rb
Facter.add('cluster_name', :sleep, :timeout => 5) do
  setcode do
    Facter::Core::Execution.exec("/bin/hostname -s | /usr/bin/sed -e 's/\d//g'")
  end
end
```

Best Practice: Always define a timeout for any fact which calls a program or connects to a dependency, to avoid hanging the Puppet client during a run.

Confining Facts

Some facts are inappropriate for certain systems, would not work or simply wouldn't provide any useful information. To limit which nodes attempt to execute the fact code, utilize a `confine` statement. This statement lists another fact name and valid values. For example, to ensure that only hosts in a certain domain provide this fact, you could use the following:

```
# cluster_name.rb
Facter.add('cluster_name') do
  confine 'domain' => 'example.com'
  setcode do
ruby code which provides the value...
```

You can use multiple `confine` statements to enforce multiple conditions, all of which must be true. Finally, you can also test against multiple values by providing an array of values. If you are looking for a fact only available on Debian-based systems, you could use this:

```
# debian_fact.rb
Facter.add('debian_fact') do
  confine 'operatingsystem' => %w{ Debian Ubuntu }
  setcode do
ruby code which provides the value...
```

Ordering by Precendence

You can define multiple methods, or *resolutions*, to acquire a fact's value. Facter will utilize the highest precendence resolution which returns a value. To provide multiple resolutions, simply add another `Facter` code block with the same fact name.

This is a common technique used for facts where the source of data is different on each operating system.

The order in which Facter evaluates possible resolutions is as follows:

1. Facter discards any resolutions where the `confine` statements do not match.
2. Facter tries each possible resolution, starting with the highest weight and descending.
3. Whenever a value is found, no further code blocks are executed.

You can define a weight for a resolution using the `has_weight` statement. If no weight is defined, the weight is equal to the number of `confine` statements in the block. This ensures that more specific resolutions are tried first.

Below is a sample definition where we try to acquire the hostname written to the system configuration files using two different locations.

```
# configured_hostname.rb
Facter.add('configured_hostname') do
  has_weight 10
  setcode do
    if File.exist? '/etc/hostname'
      File.open('/etc/hostname') do |fh|
        return fh.gets
      end
    end
```

```
  end
end

Facter.add('configured_hostname') do
  confine "os['family']" => 'RedHat'
  has_weight 5
  setcode do
    if File.exist? '/etc/sysconfig/network'
      File.open('/etc/sysconfig/network').each do |line|
        if line.match(/^HOSTNAME=(.*)$/)
          return line.match(/^HOSTNAME=(.*)$/)[0]
        end
      end
    end
  end
end
```

Aggregating Results

The data provided by a fact could be created from multiple data sources. You can then aggregate the results from all data sources together into a single result.

An aggregate fact is defined very differently than a normal fact.

1. `Facter.add()` must be invoked with a property `:type => :aggregrate`
2. Each discrete data source is defined in a `chunk` code block.
3. The `chunks` object will contain the results of all chunks.
4. An `aggregate` code block should evaluate chunks to provide the final fact value (instead of `setcode`).

Here's a simple prototype of an aggregate fact that returns an array of results.

```
Facter.add('fact_name', :type => :aggregate) do
  chunk('any-name-one') do
    ruby code
  end

  chunk('any-name-two') do
    different ruby code
  end

  aggregate do |chunks|
    results = Array.new
    chunks.each_value do |partial|
      results.push(partial)
    end
    return results
  end
end
```

The `aggregate` block is optional if all of the chucks return arrays or hashes. Facter will automatically merge the results into a single array or hash in that case. If you want a different resolution, you should define the `aggregate` block.

For more examples of aggregate resolutions, see the aggregate resolutions section of the Fact Overview page.

Best Practice: Whenever possible have multiple discrete data sources provide values in their own facts, rather than creating large, complex facts with the aggregate function.

Understanding Implementation Issues

There are a few things to understand about how the Puppet agent implements facts:

- External facts are evaluated first, and thus cannot reference or use Facter or Ruby facts.
- Ruby facts are evaluated later and can use values from External facts.
- External executable facts are forked instead of executed within the same process. This can have performance implications if thousands of external fact programs are used.

Outside of these considerations, the facts created by these programs are equal and indistinguishable.

Defining Functions

You can create your own functions to extend and enhance the Puppet language. These functions will be executed during compilation of the manifests. Functions can be written in either the Puppet Configuration Language, or in pure Ruby.

Let's cover how to define functions in either language.

Puppet Functions

New to Puppet 4 is the fully-enabled ability to write functions in the Puppet Configuration Language. This gives you the freedom to write comprehensive, powerful functions in the Puppet language without learning Ruby.

Each function should be declared in a separate file, stored in the *functions* directory of the module, and named for the function followed by the *.pp* extension.

For an example, we're going to create a function `make_boolean` which accepts many types of input and returns a boolean value. This will allow easy string (yes/no/on/off) and number (0/1) conversion to boolean. So our function would be placed in a file named *modules/puppet/functions/make_boolean.pp*.

Each function is named within the scope of the module. For our example, the function named above would be declared like so:

```
# functions/make_boolean.pp
function puppet::make_boolean( Variant[String,Numeric,Boolean] $inputvalue ) {
  if ($inputvalue =~ Numeric) {
    case $inputvalue {
      $inputval == 0: { false }
      default:        { true }
    }
  }
  elsif ($inputvalue =~ String) {
    case $inputvalue {
      /^(?-i:on|true|yes)$/ : { true }
      /^(?-i:off|false|no)$/: { false }
      default: { fail("Cannot convert '${inputvalue}' to a boolean value.") }
  }
  elsif ($inputvalue =~ Boolean) {
    $inputvalue
  }
}
```

Function alternates can be defined to expect certain types of data. This can simplify your Puppet code by skipping type checks.

```
# functions/make_boolean.pp
function puppet::make_boolean( Integer $inputvalue ) {
  case $inputvalue {
    $inputval == 0: { false }
    default:        { true }
  }
}
function puppet::make_boolean( String $inputvalue ) {
  case $inputvalue {
    /^(?-i:on|true|yes)$/ : { true }
    /^(?-i:off|false|no)$/: { false }
    default: { fail("Cannot convert '${inputvalue}' to a boolean value.") }
}
```

Ruby Functions

Each ruby function should be declared in a separate file, stored in the *lib/puppet/functions/modulename/* directory of the module, and named for the function followed by the *.rb* extension.

Define the function by calling `Puppet::Functions.create()` with the following elements:

1. The name of the new function as a Ruby Symbol as the first parameter to the function.
2. A `:type` parameter which defines whether the function returns a value or not.
3. A variable named between | pipe delimitors to accept arguments passed
4. Ruby code to implement the function within a block.

The name and arguments parameter are required, even if the function ignores the input parameters. The value for type should be either `:statement` for functions which don't return a result, or `:rvalue` for functions which do return a value.

Our `make_boolean()` function from the previous section would look like this:

```
Puppet::Functions.create(:'puppet::make_boolean', :type => :rvalue) do
  def make_boolean( value )
    if value.is_a? Integer
      return value == 0 ? false : true
    elsif value.is_a? String
      case value
      when nil or 0 or ''
        return false
      when 'false' or 'no' or 'off'
        return false
      else
        return true
      end
    end
  end
end
```

You can perform type validation using a `dispatch` block:

```
Puppet::Functions.create(:'puppet::make_boolean', :type => :rvalue) do
  dispatch :make_boolean do
    param 'String', :value
  end
  ...function definition...
end
```

Ruby functions support multiple dispatch. Set up each dispatcher with different valid input types. The dispatcher will select the first matching type and call the named function.

```
Puppet::Functions.create(:'puppet::make_boolean', :type => :rvalue) do
  dispatch :make_boolean_from_string do
    param 'String', :value
  end
```

```
  dispatch :make_boolean_from_integer do
    param 'String', :value
  end

  def make_boolean_from_integer( value )
    return value == 0 ? false : true
  end

  def make_boolean_from_string( value )
    case value
    when nil or 0 or ''
      return false
    when 'false' or 'no' or 'off'
      return false
    else
      return true
    end
  end
end
```

It's possible to accept a range or unlimited values as well. Here are dispatchers for when 2 values are supplied, and for all other amounts (e.g. unlimited) values:

```
Puppet::Functions.create(:'puppet::find_largest', :type => :rvalue) do
  dispatch :compare_two_values do
    required_param 'Integer', :first
    optional_param 'Integer', :second
    arg_count 1, 2
  end

  dispatch :compare_unlimited_values do
    repeated_param 'Integer', :values
  end

  def compare_two_values( first, second )
    ...
  end

  def compare_unlimited_values( *values )
    ...
  end
```

More examples of Ruby functions can be found at Custom Functions.

Accessing Facts and Values

Facts about the node or variables from Puppet classes can be accessed from within Ruby functions using `lookupvar()`. If the fact or variable does not exist, this function will return `nil`.

```
require 'ipaddr'
Puppet::Functions.create(:'mymodule::get_subnet', :type => :rvalue)
```

```
  def get_subnet
    ipaddress = lookupvar('facts['ipaddress']')
    if !ipaddress.nil?
      ip = IPAddr.new( ipaddress )
      return ip.mask( lookupvar('facts[netmask']') )
  end
end
```

Calling Other Functions

You can call any Ruby function or method within your custom function as documented in any Ruby reference or documentation.

You can invoke a custom Puppet function from another custom Puppet function by prepending `function_` to the name of the custom function. This prefix causes Puppet to scan all custom function paths to find and load the other function.

When a function is called from a Puppet class, all arguments are passed in as an anonymous array. If you call this function from within Ruby, you'll need to send your input as a single array to match.

```
Puppet::Functions.create(:'mymodule::outer_function')
  def outer_function
    input_array = [ ['hostname'], lookupvar('facts['hostname']') ]
    function_inner_function( input_array )
    return do_something( results )
  end
end
```

Sending Back Errors

To send an error response back to Puppet (which will cause the Puppet compilation to fail), `raise` an error of type `Puppet::ParseError`. Here's an example:

```
Puppet::Functions.create(:'mymodule::outer_function', :type => :rvalue)

  def outer_function
    raise Puppet::ParseError, 'Fact not available!' if lookupvar('facts['my_fact']').nil?
    ...things you do if the fact is available...
  end
end
```

Whenever possible it is preferred to simply return nil or some other failure value when a function doesn't succeed. This allows the code which called the function to determine what action to take. This is generally better practice than causing the entire Puppet run to fail.

Using Custom Functions

Whether your function was written in Puppet or Ruby, you can use a function you've created exactly the same as a function built into Puppet. For example, we could use the `make_boolean` function we've defined to ensure that a Service receives a boolean value, no matter what type of value was passed to it.

```
service { 'puppet':
  ensure    => $status,
  enable    => puppet::make_boolean( $enabled ),
  subscribe => Package['puppet-agent'],
}
```

To call a custom function within a Puppet Template, you need to use the same `function_` prefix as used within a Ruby function. This is because, as you might guess, templates are parsed within the scope of a Ruby function. Use brackets around the input variables to create a single array for input.

```
<%= if scope.function_puppet::make_boolean( [ somevalue ] ) -%>
```

Providing Data in Modules

There are plans to provide for data in Modules with Puppet 4. This did not make the 4.0 or 4.1 releases, and plans for this feature are still developing. I will update this section as more information becomes available.

Module Plugins Review

In this chapter we have covered how to extend a module to:

- Provide new Facts that will be available for any module to reference.
  - Facts can be written in Ruby, and use Ruby language features.
  - Facts can be read from YAML, JSON, or text file formats
  - Facts can be read from any executible program or script that outputs one fact name and value per line.
- Provide new Functions that will be available for any module to reference.
  - Facts can be written in the Puppet language and use all Puppet features.
  - Facts can be written in Ruby, and use all Ruby language features.

New features for Data in Modules are being worked on, and will be documented in the next update to this book.

Requirements for Module Plugins

There were a lot of rules around how module artifacts are named and created. Let's go over the rules we covered in this section:

- External fact programs should be placed in the *facts.d/* directory and be executible by the Puppet user. Windows fact providers need to be named with a known file extension.
- External fact data should be placed in the *facts.d/* directory and have a file extension of *.yaml*, *.json*, or *.txt*.
- Functions written with the Puppet language should be placed in the *functions/* directory and be named with a *.pp* file extension.
- Ruby functions should be placed in the *lib/puppet/functions/`modulename`/* directory and be named the same as the function with a *.rb* file extension.
- Ruby functions or templates which call custom functions need to prefix the function name with `function_`.
- Ruby functions or templates which call custom functions need to pass all input parameters in a single array.

You can find more detailed guidelines in the Puppet Labs Style Guide.

CHAPTER 16

Publishing Modules

This chapter will cover how to share your module with others, both inside and outside of your organization.

Packaging a Module

To share your module you must prepare the module, then package it for upload.

First, review your Module Documentation and make sure it is up to date. In particular, make sure that the *README.md* and *CHANGELOG.md* files are up to date.

Next, edit the *metadata.json* file to indicate the license, where to find the source code, and where to report issues. Here is an example file:

```
$ cat metadata.json
{
  "name": "jorhett-puppet",
  "version": "1.2.1",
  "author": "Jo Rhett",
  "summary": "Module to configure and manage Puppet agent, master, and server.",
  "license": "BSD-3-Clause",
  "source": "https://github.com/jorhett/puppet",
  "project_page": 'https://github.com/jorhett/puppet',
  "issues_url": 'https://github.com/jorhett/puppet/issues',
```

Within the same file, indicate the operating systems and Puppet versions supported. Note that `operatingsystem_support` is a hash, while `requirements` and `dependencies` are arrays.

```
  "operatingsystem_support": {
    {
      "operatingsystem":"RedHat",
      "operatingsystemrelease":[ "5", "6", "7" ]
    },
```

```
    {
      "operatingsystem": "Ubuntu",
      "operatingsystemrelease": [ "15.04", "14.04", "14.10", "13.04", "13.10" ]
    }
  },
  "requirements": [
    {
      "name": "pe",
      "version_requirement": "3.2.x"
    },
    {
      "name": "puppet",
      "version_requirement": [ "4.x", "3.x" ]
    }
  ],
  "dependencies": [
    {"version_requirement":">= 4.0.0","name":"puppetlabs-stdlib"}
    {"version_requirement":">= 1.0.0","name":"puppetlabs-inifile"}
  ]
}
```

Finally, use the `puppet module build` command to package up your module.

```
$ puppet module build
Notice: Building jorhett-puppet for release
Module built: jorhett-puppet/pkg/jorhett-puppet-1.2.1.tar.gz
```

Uploading a Module to the Puppet Forge

There is no API for uploading modules to the Puppet Forge. You do this manually using a web browser. the process for uploading a new module and upgrading one of your existing modules is identical.

1. Navigate to http://forge.puppetlabs.com/
2. Click *Sign up* if necessary to create a new account
3. Click *Login* if you already have an account
4. Click on *Publish* in the upper right hand corner of the screen underneath your name
5. Click *Choose File* and select the module package you created in the previous step
6. Click *Upload*

Your module will be added to the Forge. The Readme and ChangeLog from your module will be shown as the web page for your module. The results of standard tests and community feedback will be added to the page when they are available.

If this process has changed since this book was published, the revised process should be documented at Publishing Modules on the Puppet Forge.

Publishing a Module on GitHub

It is common and expected for you to have a place to accept bug reports and pull requests for your module. GitHub is by far the most common location to do this. The following steps will create a GitHub repository for your module.

If you haven't installed git already, you should do that now.

```
[vagrant@client ~]$ sudo yum install -y git
...snip...
Installed:
  git.x86_64 0:1.8.3.1-4.el7

Dependency Installed:
  libgnome-keyring.x86_64 0:3.8.0-3.el7       perl-Error.noarch 1:0.17020-2.el7       perl-Git.n

Complete!
```

Configure the Git software with your name and e-mail:

```
$ git config --global user.name "Jane Doe"
$ git config --global user.email janedoe@example.com
```

Next, create a GitHub account if you don't already have one at https://github.com/join.

Create a new repository in your GitHub Account at https://github.com/new. You will likely want to name the repository by replacing your name in the module with *puppet-* as GitHub already organizes the repository under your name. Ignore the options below the name and click *Create Repository*.

Setup version tracking of your module by running the following command within your module directory.

```
$ git init
Initialized empty Git repository in
    /home/vagrant/.puppet/modules/jorhett-puppet/.git/
```

There are some files to avoid uploading to the source repository. These include:

- Binary packages of the module
- Dependency fixtures created by rspec for testing

To prevent this, create a *.gitignore* file with the following contents:

```
# .gitignore
/pkg/
/spec/fixtures/
```

Commit your module to the Git repository by running the following commands within your module directory:

```
$ git add --all

$ git commit -m "Initial commit"
[master (root-commit) e804295] initial commit
11 files changed, 197 insertions(+), 0 deletions(-)
 create mode 100644 Gemfile
 create mode 100644 README.md
 create mode 100644 CHANGELOG.md
 create mode 100644 Rakefile
 create mode 100644 manifests/init.pp
 create mode 100644 manifests/client.pp
 create mode 100644 manifests/master.pp
 create mode 100644 metadata.json
 create mode 100644 spec/classes/init_spec.rb
 create mode 100644 spec/spec_helper.rb
 create mode 100644 tests/init.pp
```

You have now committed your changes to the source tree. However they are not yet pushed up to GitHub. Let's configure GitHub as our remote origin. *If you have a different origin and are just pushing to GitHub as a remote branch, we expect that you know how to do that.*

```
$ git remote add origin https://github.com/janedoe/modulename.git
$ git push -u origin master
Counting objects: 14, done.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (14/14), 3.68 KiB | 0 bytes/s, done.
Total 14 (delta 0), reused 0 (delta 0)
To https://github.com/jorhett/puppet-systemstd.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Whenever you wish to publish a change to the module, make sure to update the version number and changes for the version in the following files.

- metadata.json
- README.md
- CHANGELOG.md

Commit these changes to the repository, and then push them up to GitHub.

```
$ git commit -a -m "Updated documentation for version X.Y"
[master a4cc6b7] Updated documentation
 Committer: jorhett vagrant@client.example.com
 3 files changed, 11 insertions(+), 2 deletions(-)
$ git push
Counting objects: 14, done.
```

```
Compressing objects: 100% (11/11), done.
...
```

Automating Module Publishing

There is a community-provided Ruby gem which automates the task of updating your module on the Forge. You can find documentation for this at https://github.com/maestrodev/puppet-blacksmith.

I don't recommend using the version `bump` or all-in-one `release` features since you will need to manually update the *CHANGELOG.md* file with the version changes anyway. However if you really hate using a browser, it allows you to automate the upload with a single command from within the module directory:

```
$ rake module:push
```

This involves adding two lines to the Rakefile in the Puppet module directory, and then using the following commands to

Getting Approved Status from Puppet Labs

At the time this book was written the requirements for Approved status were not considered stable, and were likely to evolve. At this time the requirements were as follows:

- Solve a unique problem well. Puppet Labs won't approve multiple modules which solve the same problem.
- Compliance with Puppet Style Guide. No warnings issued by syntax checking tools like `puppet-lint`.
- Regularly updated by more than one person or organization. Having Forge updates in the last 6 months, and less than 1 month lag between source repo (e.g. GitHub) and the Forge.
- Thorough and readable documentation.
- Licensed under Apache, MIT, or BSD licenses.
- Have every standard metadata field filled out, including Puppet version and operating system compatibility.
- Versioned according to SemVer expectations to keep expectations consistent with regards to upgrades.
- Should have rspec and acceptance tests for every manifest, and unit tests for types, providers, facts, and functions.

Given the nature of supporting a community of published modules, I feel the requirements are straightforward, easy to understand, and generally easy to apply.

You can check for updates to the approval guidelines at https://forge.puppetlabs.com/approved/criteria.