



Community Experience Distilled

Extending Puppet

Second Edition

Start pulling the strings of your infrastructure with Puppet – learn how to configure, customize, and manage your systems more intelligently

Alessandro Franceschi
Jaime Soriano Pastor

[PACKT] open source*
PUBLISHING community experience distilled

Extending Puppet

Second Edition

Start pulling the strings of your infrastructure with Puppet – learn how to configure, customize, and manage your systems more intelligently

Alessandro Franceschi

Jaime Soriano Pastor

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Extending Puppet

Second Edition

Copyright © 2016 Packt Publishing

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

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

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

First published: June 2016

Production reference: 1170616

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

ISBN 978-1-78588-568-6

www.packtpub.com

Credits

Authors

Alessandro Franceschi
Jaime Soriano Pastor

Reviewer

Jeroen Hooyberghs

Commissioning Editor

Priya Singh

Acquisition Editors

Chaitanya Nair
Shaon Basu

Content Development Editor

Merint Thomas Mathew

Technical Editor

Deepti Tuscano

Copy Editor

Merilyn Pereira

Project Coordinator

Francina Pinto

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Authors

Alessandro Franceschi is a long time Puppet user, trainer, and consultant.

He started using Puppet in 2007, automating a remarkable amount of customers' infrastructures of different sizes, natures, and complexities.

He has attended several PuppetConf and PuppetCamps as both speaker and participant, always enjoying the vibrant and friendly community, each time learning something new.

Over the years, he started to publish his Puppet code, trying to make it reusable in different scenarios. The result of this work is the example42 Puppet modules and control repo, complete, feature rich, sample Puppet environment. You can read about example42 at www.example42.com.

You can follow Franceschi on his Twitter account at [@alvagante](https://twitter.com/alvagante).

Jaime Soriano Pastor was born in Teruel, a small city in Spain. He has always been a passionate about technology and sciences. While studying computer science at the university in his hometown, he had his first encounter with Linux and free software, which is what shaped his career.

He worked for several companies on different and interesting projects, from operating systems in embedded devices to the cloud, giving him a wide view on several fields of software development and systems administration.

Currently, automation, configuration management, and continuous integration form a part of his daily work in the SRE team at Tuenti Technologies.

About the Reviewer

Jeroen Hooyberghs is an open source and Linux enthusiast, available as consultant at Open-Future in Belgium. In this position, as well as in previous jobs in Linux system administration, he built his technical expertise in a lot of open source solutions. In the last couple of years he's been focusing on system automization and configuration management, mainly using Puppet. Since 2014, he's been a Puppet Certified Professional and Puppet trainer.

As a technical reviewer, he contributed to other Packt Publishing books, such as *Mastering Puppet* by Thomas Uphill and *Puppet Cookbook* by John Arundel and Thomas Uphill.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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



<https://www2.packtpub.com/books/subscription/packtlib>

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

Why subscribe?

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

Free access for Packt account holders

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

Table of Contents

Preface	ix
Chapter 1: Puppet Essentials	1
The Puppet ecosystem	2
Why configuration management matters	3
Puppet components	4
Installing and configuring Puppet	5
Puppet in action	8
Resources	9
The resource abstraction layer	10
Nodes	12
Classes and defines	13
Class inheritance	16
Resource defaults	17
Resource references	17
Variables, facts, and scopes	18
System's facts	18
External facts	19
User variables in Puppet DSL	20
Type-checking	21
User variables in an ENC	22
User variables in Hiera	22
Puppet built-in variables	23
Variables scope	24
Meta parameters	25
Managing order and dependencies	25
Run stages	27
Reserved names and allowed characters	28
Conditionals	28

Comparison operators	29
Iteration and lambdas	30
The in operator	31
Expressions combinations	31
Exported resources	31
Virtual resources	32
Modules	33
Paths of a module and auto loading	33
Templates	35
Restoring files from a filebucket	36
Summary	38
Chapter 2: Managing Puppet Data with Hiera	39
Installing and configuring Hiera	40
The hiera.yaml configuration file	41
Global settings	42
Backend specific settings	42
Examples	42
Working with the command line on a YAML backend	44
Using Hiera in Puppet	49
Dealing with hashes in Puppet code	50
Puppet 3 automatic parameter lookup	51
Evolving usage patterns for class parameters	52
Additional Hiera backends	53
Hiera-file	53
Encryption plugins	55
Hiera-gpg	55
Hiera-eyaml	57
Hiera-http, hiera-mysql	59
Using Hiera as an ENC	60
Summary	61
Chapter 3: Introducing PuppetDB	63
Installation and configuration	64
Installing PuppetDB	65
PuppetDB configurations	67
Init script configuration	68
Configuration settings	68
Logging configuration	73
Configurations on the Puppet Master	73
Dashboards	74
PuppetDB performance dashboard	74
Puppetboard—Querying PuppetDB from the Web	76

PuppetDB API	76
Querying PuppetDB (read)	76
PuppetDB commands (write)	79
Querying PuppetDB for fun and profit	80
/facts endpoint	81
/resources endpoint	81
/nodes endpoint	82
/catalogs endpoint	83
/fact-names endpoint	83
/metrics endpoint	83
/reports endpoint	84
/events endpoint	84
/event-counts endpoint	85
/aggregated-event-counts endpoint	86
/server-time endpoint	86
/version endpoint	86
The puppetdbquery module	86
Query format	87
Querying from the command line	87
Querying from Puppet manifests	88
The PuppetDB Hiera backend	88
How Puppet code may change in the future	89
Summary	90
Chapter 4: Designing Puppet Architectures	91
Components of a Puppet architecture	92
Definition of the classes to include in each node	94
Definition of the parameters to use for each node	95
Definition of the configuration files provided to the nodes	98
Definition of custom resources and classes	99
The Foreman	99
Roles and profiles	101
The data and the code	103
Sample architectures	104
The default approach	105
Basic ENC, logic on site module, Hiera backend	108
The Foreman and Hiera	109
Hiera-based setup	110
Foreman smart variables	111
Facts-driven truths	112
Nodeless site.pp	113
Summary	114

Chapter 5: Using and Writing Reusable Modules	115
Modules layout evolution	116
Class parameters—from zero to data bindings	116
Params pattern	117
Data in modules	119
Files and class names	120
The anchor pattern	121
The parameters dilemma	122
Naming standards	124
Reusability patterns	125
Managing files	126
Managing configuration hash patterns	128
Managing multiple configuration files	130
Managing users and dependencies	132
Managing installation options	133
Managing extra resources	134
Summary	136
Chapter 6: Higher Abstraction Modules	137
The OpenStack example	139
Component (application) modules	139
Raising abstraction – the official OpenStack module	140
Raising abstraction – Puppet Labs OpenStack module	141
Taking an alternate approach	143
An approach to reusable stack modules	144
Tiny Puppet	149
Summary	151
Chapter 7: Puppet Migration Patterns	153
Examining potential scenarios and approaches	153
New infrastructures	154
Existing manually managed infrastructures	155
Node migration	156
Node update	157
Existing automated infrastructures	158
Patterns for extending Puppet coverage	160
Raising the bar, step by step	160
Knowing the territory	161
Defining priorities	162
Evaluating solutions	166
Coding	167
Applying changes	170

Things change	171
Puppet friendly infrastructure	174
Puppet-friendly software	175
Summary	177
Chapter 8: Code Workflow Management	179
Write Puppet code	179
Geppetto	180
Vim	181
Git workflows	182
Code management using Git	182
Git hooks	183
Environments and branches	184
Branch based automatic environments	185
Simplified developer workdir environments	185
Code review	186
Gerrit	186
Online resources for peer review	187
Testing Puppet code	187
Using rspec-puppet	188
Vagrant	190
Beaker and beaker-rspec	192
Deploying Puppet code	194
Using librarian-puppet for deployments	195
Deploying code with r10k	196
Propagating Puppet changes	196
Puppet continuous integration	197
Travis	198
Jenkins	199
Summary	200
Chapter 9: Scaling Puppet Infrastructures	201
Scaling Puppet	202
As simple as puppet apply	202
Default Puppet Master	203
Puppet Master with Passenger	204
Installing and configuring Passenger	205
Puppet Master based on Trapperkeeper	209
Multi-Master scaling	210
Managing certificates	211
Managing SSL termination	213
Managing code and data	215

Load balancing alternatives	215
Masterless Puppet	216
Configurations and infrastructure optimizations	217
Traffic compression	217
Caching	218
Distributing Puppet run times	218
File system changes check	219
Scaling stored configs	219
Measuring performance	221
Puppet Metrics	222
Optimizing code	223
Testing different Puppet versions	225
Summary	226
Chapter 10: Extending Puppet	227
Anatomy of a Puppet run, under the hood	228
Puppet extension alternatives	230
ENC and Hiera Extendability	230
Modules pluginsync	231
Puppet indirector and its termini	232
Custom functions	236
Custom facts	239
Ruby facts distributed via pluginsync	240
External facts in facts.d directory	242
Custom types and providers	243
Custom report handlers	250
Custom faces	252
Summary	256
Chapter 11: Beyond the System	257
Puppet on network equipment	257
Proxy mode with Puppet device application	259
Native Puppet on network equipment	261
Cisco onePK	262
Juniper and the netdev_stdlib	263
Puppet for cloud and virtualization	264
VMware	265
VM provisioning on vCenter and vSphere	265
vCenter configuration	265
vSphere virtual machine management with resource types	266
Amazon Web Services	267
Cloud provisioning on AWS	267

AWS provisioning and configuration with resource types	269
Managing CloudFormation	270
Cloud provisioning on Google Compute Engine	270
Puppet on storage devices	271
Puppet and Docker	271
Summary	274
Chapter 12: Future Puppet	275
Changing the serialization format	276
Direct Puppet	278
File sync	279
Other changes	279
Certificate authority in Clojure	280
Package management	280
Changes in faces	281
Beyond Puppet 4.x	281
Summary	282
Index	285

Preface

Puppet has changed the way we manage our systems.

When it was released, other configuration management tools were around, but it was clear that it had something special. It came at the right time with the right approach. The challenges of IT infrastructures were beginning to step up to a new level, and the need to automate common activities such as a quick setup and configuration of systems was becoming a requirement. Puppet presented a sane model, based on the abstraction of resources and the definition of the expected state of a system, using a clear and sysadmin-friendly language.

There are various books about Puppet around, and most of them are very good. This one tries to contribute with solid, no frills content (few pictures, few large blocks of copied and pasted text) and some new perspectives and topics. It begins with an intense technical overview of Puppet, Hiera, and PuppetDB, so that you can use them to design appropriate Puppet architectures that fit your IT infrastructure.

We will explore where our data can be placed, how to design reusable modules, and how they can be used as building blocks for higher abstraction classes. We will try to give a clearer and wider view of what it means to work with Puppet and what are the challenges to face when we introduce it in our systems, from code management to deployment rollouts. We will dive into Puppet's internals and its extension points, showing the multiple ways we can tweak, extend, and hack with it. We will also look at less traditional fields, such as Puppet as a configuration management tool for network devices or cloud services.

Since the first edition of this book was published by Alessandro in 2014, it has been a reference for both learners that wanted to start with Puppet and for experienced users that were looking for a deeper understanding of this tool. During these years, some things have changed in Puppet and the content of the book needed to be refreshed. Probably the most important release of Puppet till now, version 4, happened and it introduced some changes in several aspects. This edition covers the main differences that appeared with this new version, how it is installed, compatibility between versions, and so on. It also adds descriptions for the new language features so it can continue being used as a reference for learners of all levels. Some of the contents that may have become obsolete have been updated or replaced by new implementations or better practices. Some sections have been extended to provide further details on things that have proven to be of interest to the community. And in the last chapter, we will also analyze some of the steps Puppet is taking or will take in the future.

What this book covers

Chapter 1, Puppet Essentials, is an intense and condensed summary of the most important Puppet concepts; it acts as the baseline needed to understand the chapters that follow and a good occasion to refresh and maybe enrich knowledge about the Puppet language and model.

Chapter 2, Managing Puppet Data with Hiera, is dedicated to how to manage our data with Hiera, define the lookup hierarchy, organize data, and use different backends.

Chapter 3, Introducing PuppetDB, covers the installation, configuration, and usage of PuppetDB, and explores the great possibilities that it may enable in the next generations of modules.

Chapter 4, Designing Puppet Architectures, outlines the components to manage when defining a Puppet architecture: the available tools, how to integrate them, how to cope with data and code, and how to organize resources to be applied to nodes.

Chapter 5, Using and Writing Reusable Modules, covers the most important Puppet element from the user's perspective: modules and how to write them in order to be able to reuse them in different infrastructures.

Chapter 6, Higher Abstraction Modules, takes a step further and focuses on modules that use different application modules to compose more complex and wider scenarios.

Chapter 7, Puppet Migration Patterns, analyzes the approaches that can be taken when introducing Puppet in a new or existing infrastructure: methodologies, patterns, techniques, and tips for a successful deployment.

Chapter 8, Code Workflow Management, focuses on how to manage Puppet code, from when is written in an editor, to its management with an SCM, its testing and deployment, and into production.

Chapter 9, Scaling Puppet Infrastructures, faces the challenges in growing infrastructures and how it is possible to make Puppet scale with them.

Chapter 10, Extending Puppet, covers the many available possibilities to extend the core code with custom plugins and gives a deeper view of how Puppet's internals are organized.

Chapter 11, Beyond the System, takes a journey outside the traditional territories, exploring how we can manage with the Puppet network, as well as storage equipment and cloud instances.

Chapter 12, Future Puppet, is a step beyond Puppet 4 and how the project may evolve in future versions.

What you need for this book

You can test the Puppet code present in this book on any Linux system connected to the Internet.

Who this book is for

This book is accessible to any Puppet user.

If you are totally new to Puppet, be sure to thoroughly read *Chapter 1, Puppet Essentials*, and have properly understood its principles before continuing.

If you are an intermediate user, it is recommended that you read the chapters in order.

If you are an advanced user, you may pick from different pages useful information and new insights on topics you should already know.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "When the Master receives a connection then it looks in its manifests (starting from `/etc/puppet/manifests/site.pp`) what resources have to be applied for that client host, also called node."

A block of code is set as follows:

```
class my_users {
  @user { 'al': [...] tag => 'admins' }
  @user { 'matt': [...] tag => 'developers' }
  @user { 'joe': [...] tag => 'admins' }
  [ ... ]
}
```

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

```
ls -l $(factor rubysitedir)/puppet/provider/package/
```

 [Warnings or important notes appear in a box like this.]

 [Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

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

Customer support

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

Errata

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

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

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

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

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

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Puppet Essentials

There are moments in our professional life when we meet technologies that trigger an inner *wow* effect. We realize there's something special in them and we start to wonder how they can be useful for our current needs and, eventually, wider projects. Puppet, for me, has been one of these turning point technologies. I have reason to think that we might share a similar feeling.

If you are new to Puppet, you are probably starting from the wrong place, there are better fitting titles around to grasp its basic concepts.

This book won't indulge too much on the fundamentals, but don't despair, this chapter might help for a quick start. It provides the basic Puppet background needed to understand the rest of the contents and may also offer valuable information to more experienced users.

We are going to review the following topics:

- The Puppet ecosystem, its components, history, and the basic concepts behind configuration management
- How to install and configure Puppet commands and paths, to understand where things are placed
- The core components and concepts; terms such as manifests, resources, nodes, and classes will become familiar
- The main language elements – variables, references, resources defaults, ordering, conditionals, comparison operators, virtual and exported resources
- How Puppet stores the changes it makes and how to revert them

The contents of this chapter are quite dense, so take your time to review and assimilate them if they sound new or look too complex; the path towards Puppet awareness is never too easy.

The Puppet ecosystem

Puppet is a configuration management and automation tool; we use it to install, configure, and manage components of our servers.

Initially written in **Ruby**, some parts were rewritten in version 4 in **Clojure**. Released with an **open source** license (Apache 2), it can run on any Linux distribution, many other UNIX variants (Solaris, *BSD, AIX, and Mac OS X), and Windows. Its development started in 2005 by Luke Kanies as an alternative approach to the existing configuration management tools (most notably CFEngine and BladeLogic). The project has grown year after year; Kanies' own company, Reductive Labs, renamed in 2010 to Puppet Labs, has received a total funding of \$ 45.5 million in various funding rounds (among the investors there are names such as VMware, Google, and Cisco).

Now, it is one of the top 100 fastest growing companies in the US. It employs more than 150 people and it has a solid business based on open source software, consisting of consulting services, training, certifications, and **Puppet Enterprise**. Puppet Enterprise is the commercial version that is based on the same open source Puppet codebase, but it provides an integrated stack with lots of tools, such as a web GUI that improves and makes Puppet usage and administration easier, and more complete support for some major Linux distributions, Mac OS X, and Microsoft Windows Server.

The Puppet ecosystem features a vibrant, large, and active community, which discusses on the Puppet Users and Puppet Developers Google groups, on the crowded free node #puppet IRC channel, at the various Puppet Camps that are held multiple times a year all over the world, and at the annual PuppetConf, which is improving and getting bigger year after year.

Various software products are complementary to Puppet; some of them are developed by Puppet Labs:

- **Hiera**: This is a key-value lookup tool that is the current choice of reference for storing data related to our Puppet infrastructure.
- **Mcollective**: This is an orchestration framework that allows parallel execution of tasks on multiple servers. It is a separate project by Puppet Labs, which works well with Puppet.
- **Facter**: This is a required complementary tool; it is executed on each managed node and gathers local information in key/value pairs (facts), which are used by Puppet.
- **Geppetto**: This is an IDE, based on Eclipse that allows easier and assisted development of Puppet code.

- **Puppet Dashboard:** This is an open source web console for Puppet.
- **PuppetDB:** This is a powerful backend that can store all the data gathered and generated by Puppet.
- **Puppet Enterprise:** This is the commercial solution to manage via a web frontend Puppet, Mcollective, and PuppetDB.

The community has produced other tools and resources. The most noticeable ones are:

- **The Foreman:** This is a systems lifecycle management tool that integrates perfectly with Puppet.
- **PuppetBoard:** This is a web front end for PuppetDB.
- **Kermit:** This is a web front end for Puppet and Mcollective.
- **Modules:** These are reusable components that allow management of any kind of application and software via Puppet.

Why configuration management matters

IT operations have changed drastically in the past few years. Virtualization, cloud, business needs, and emerging technologies have accelerated the pace of how systems are provisioned, configured, and managed.

The manual setup of a growing number of operating systems is no longer a sustainable option. At the same time, in-house custom solutions to automate the installation and the management of systems cannot scale in terms of required maintenance and development efforts.

For these reasons, configuration management tools such as Puppet, Chef, CFEngine, Rudder, Salt, and Ansible (to mention only the most known open source ones) are becoming increasingly popular in many infrastructures.

They show infrastructure as code, that allows, in systems management, the use of some of the same best practices in software development for decades, such as maintainability, code reusability, testability, or version control.

Once we can express the status of our infrastructure with versioned code, there are powerful benefits:

- We can **reproduce** our setups in a consistent way, what is executed once can be executed any time, the procedure to configure a server from scratch can be repeated without the risk of missing parts
- Our code commits log reflects the **history of changes** on the infrastructure; who did what, when, and if commits comments are pertinent, why.

- We can **scale** quickly; the configurations we made for a server can be applied to all the servers of the same kind.
- We have **aligned and coherent environments**; our Development, Test, QA, Staging, and Production servers can share the same setup procedures and configurations.

With these kinds of tools, we can have a system provisioned from zero to production in a few minutes, or we can quickly propagate a configuration change over our whole infrastructure automatically.

Their power is huge and has to be handled with care; as we can automate massive and parallelized setups and configurations of systems; we might automate distributed destructions.

With great power comes great responsibility.

Puppet components

Before diving into installation and configuration details, we need to clarify and explain some Puppet terminology to get the whole picture.

Puppet features a declarative **Domain Specific Language (DSL)**, which expresses the desired state and properties of the managed resources.

Resources can be any component of a system, for example, packages to install, services to start, files to manage, users to create, and also custom and specific resources, such as MySQL grants, Apache virtual hosts, and so on.

Puppet code is written in **manifests**, which are simple text files with a `.pp` extension. Resources can be grouped in **classes** (do not consider them classes as in OOP, they aren't). Classes and all the files needed to define the configurations required are generally placed in **modules**, which are directories structured in a standard way that are supposed to manage specific applications or system's features (there are modules to manage Apache, MySQL, sudo, sysctl, networking, and so on).

When Puppet is executed, it first runs **facter**, a companion application, which gathers a series of variables about the system (IP address, hostname, operating system, and MAC address), which are called **facts** and are sent to the Master.

Facts and user-defined **variables** can be used in manifests to manage how and what resources to provide to clients.

When the Master receives a connection, then it looks in its manifests (starting from `/etc/puppet/manifests/site.pp`) what resources have to be applied for that client host, also called **node**.

The Master parses all the DSL code and produces a **catalog**, which is sent back to the client (in PSON format, a JSON variant used in Puppet). The production of the catalog is often referred to as catalog **compilation**.

Once the client receives the catalog, it starts to apply all the resources declared there; packages are installed (or removed), services started, configuration files created or changed, and so on. The same catalog can be applied multiple times, if there are changes on a managed resource (for example, a manual modification of a configuration file) they are reverted back to the state defined by Puppet; if the system's resources are already at the desired state, nothing happens.

This property is called **idempotence** and is at the root of the Puppet declarative model; since it defines the desired state of a system, it must operate in a way that ensures that this state is obtained whatever are the starting conditions and the number of times Puppet is applied.

Puppet can report the changes it makes on the system and audit the drift between the system's state and the desired state as defined in its catalog.

Installing and configuring Puppet

Puppet uses a client-server paradigm. Clients (also called **agents**) are installed on all the systems to be managed, and the server(s) (also called **Master**) is installed on a central machine(s) from where we control the whole infrastructure.

We can find Puppet's packages on most recent OS, either in the default repositories or in other ones maintained by the distribution or its community (for example, **EPEL** for Red Hat derivatives).

Starting with Puppet version 4.0, Puppet Labs introduced **Puppet Collections**. These collections are repositories containing packages that can be used between them. When using collections, all nodes in the infrastructure should be using the same one. As a general rule, Puppet agents are compatible with newer versions of Puppet Master, but Puppet 4 breaks compatibility with previous versions.

To look for the more appropriate packages for our infrastructure, we should use Puppet Labs repositories.

The server package is called `puppetserver`, so to install it we can use these commands:

```
apt-get install puppetserver # On Debian derivatives
yum install puppetserver # On Red Hat derivatives
```

And similarly, to install the agent:

```
apt-get install puppet-agent # On Debian derivatives
```

```
yum install puppet-agent # On Red Hat derivatives
```



To install Puppet on other operating systems, check out <http://docs.puppetlabs.com/guides/installation.html>.

In versions before 4.0, the agent package was called `puppet`, and the server package was called `puppetmaster` on Debian-based distributions and `puppet-server`, on Red Hat, derivated distributions.

This will be enough to start the services with the default configuration, but the commands are not installed in any of the usual standard paths for binaries, we can find them under `/opt/puppetlabs/puppet/bin`, we'd need to add it to our `PATH` environment variable if we want to use them without having to write the full path.

Configuration files are placed in versions before 4.0 in `/etc/puppet`, and from 4.0 in `/etc/puppetlabs/` as well as the configuration of other Puppet Labs utilities as Mcollective. Inside the `puppetlabs` directory, we can find the Puppet one, that contains the `puppet.conf` file; this file is used by both agents and server, and includes the parameters for some directories used in runtime and specific information for the agent, for example, the server to be used. The file is divided in [sections] and has an INI-like format. Here is the content just installed:

```
[master]
vardir = /opt/puppetlabs/server/data/puppetserver
logdir  = /var/log/puppetlabs/puppetserver
rundir  = /var/run/puppetlabs/puppetserver
pidfile = /var/run/puppetlabs/puppetserver/puppetserver.pid
codedir = /etc/puppetlabs/code

[agent]
server = puppet
```

A very useful command to see all the current client configuration settings is:

```
puppet config print all
```

The server has additional configuration in the `puppetmaster/conf.d` directory in files in the HOCON format, a human-readable variation of JSON format. Some of these files are as follows:

- `global.conf`: This contains global settings; their defaults are usually fine
- `webserver.conf`: This contains webserver settings, such as the port and the listening address
- `puppetserver.conf`: This contains the settings used by the server
- `ca.conf`: This contains the settings for the Certificate Authority service

Configurations in previous files refer to other files on some occasions, which are also important to know when we work with Puppet:

- **Logs**: They are in `/var/log/puppetlabs` (but also on normal syslog files, with facility daemon), both for agents and servers
- **Puppet operational data**: This is placed in `/opt/puppetlabs/server/data/puppetserver`
- **SSL certificates**: They are stored in `/opt/puppetlabs/puppet/ssl`. By default, the agent tries to contact a server hostname called `puppet`, so either name our server `puppet.$domain` or provide the correct name in the server parameter.
- **Agent certificate name**: When the agent communicates with the server, it presents itself with its `certname` (is also the hostname placed in its SSL certificates). By default, the `certname` is the **fully qualified domain name (FQDN)** of the agent's system.
- **The catalog**: This is the configuration fetched by the agent from the server. By default, the agent daemon requests that every 30 minutes. Puppet code is placed, by default, under `/etc/puppetlabs/code`.
- **SSL certificate requests**: On the Master, we have to sign each client's certificates request (manually by default). If we can cope with the relevant security concerns, we may automatically sign them by adding their FQDNs (or rules matching them) to the `autosign.conf` file, for example, to automatically sign the certificates for `node.example.com` and all nodes whose hostname is a subdomain for `servers.example.com`:

```
node.example.com
*.servers.example.com
```

However, we have to take into account that any server can request configuration with any FQDN so this is potentially a security flaw.

Puppet in action

Client-server communication is done using REST-like API calls on a SSL socket, basically it's all HTTPS traffic from clients to the server's port 8140/TCP.

The first time we execute Puppet on a node, its x509 certificates are created and placed in `ssldir`, and then the Puppet Master is contacted in order to retrieve the node's catalog.

On the Puppet Master, unless we have `autosign` enabled, we must manually sign the clients' certificates using the `cert` subcommand:

```
puppet cert list # List the unsigned clients certificates
puppet cert list --all # List all certificates
puppet cert sign <certname> # Sign the given certificate
```

Once the node's certificate has been recognized as valid and been signed, a trust relationship is created and a secure client-server communication can be established.

If we happen to recreate a new machine with an existing `certname`, we have to remove the certificate of the old client from the server:

```
puppet cert clean <certname> # Remove a signed certificate
```

At times, we may also need to remove the certificates on the client; a simple move command is safe enough:

```
mv /etc/puppetlabs/puppet/ssl /etc/puppetlabs/puppet/ssl.bak
```

After that, the whole directory will be recreated with new certificates when Puppet is run again (never do this on the server — it'll remove all client certificates previously signed and the server's certificate, whose public key has been copied to all clients).

A typical Puppet run is composed of different phases. It's important to know them in order to troubleshoot problems:

1. Execute Puppet on the client. On a root shell, run:

```
puppet agent -t
```
2. If `pluginsync = true` (default from Puppet 3.0), then client retrieves any extra plugin (facts, types, and providers) present in the modules on the Master's `$modulepath` client output with the following command:

```
Info: Retrieving pluginfacts
Info: Retriving plugin
```

3. The client runs `facter` and sends its facts to the server client output:
Info: Loading facts in /var/lib/puppet/lib/facter/... [...]
4. The server looks for the client's `certname` in its nodes list.
5. The server compiles the catalog for the client using its facts. Server logs as follows:
Compiled catalog for <client> in environment production in 8.22 seconds
6. If there are syntax errors in the processed Puppet code, they are exposed here and the process terminates; otherwise, the server sends the catalog to the client in the PSON format. Client output is as follows:
Info: Caching catalog for <client>
7. The client receives the catalog and starts to apply it locally. If there are dependency loops, the catalog can't be applied and the whole run fails. Client output is as follows:
Info: Applying configuration version '1355353107'
8. All changes to the system are shown on `stdout` or in logs. If there are errors (in red or pink, depending on Puppet versions), they are relevant to specific resources but do not block the application of the other resources (unless they depend on the failed ones). At the end of the Puppet run, the client sends a report of what has been changed to the server. Client output is as follows:
Notice: Applied catalog in 13.78 seconds
9. The server sends the report to a report collector if enabled.

Resources

When dealing with Puppet's DSL, most of the time, we use resources as they are single units of configuration that express the properties of objects on the system. A resource declaration is always composed of the following parts:

- **type:** This includes package, service, file, user, mount, exec, and so on
- **title:** This is how it is called and may be referred to in other parts of the code
- Zero or more **attributes:**

```
type { 'title':
  attribute => value,
  other_attribute => value,
}
```

Inside a catalog, for a given type, there can be only one title; there cannot be multiple resources of the same type with the same title, otherwise we get an error like this:

Error: Duplicate declaration: <Type>[<name>] is already declared in file <manifest_file> at line <line_number>; cannot redeclare on node <node_name>.

Resources can be **native** (written in Ruby), or **defined** by users in Puppet DSL.

These are examples of common native resources; what they do should be quite obvious:

```
file { 'motd':  
  path    => '/etc/motd',  
  content => "Tomorrow is another day\n",  
}  
  
package { 'openssh':  
  ensure => present,  
}  
  
service { 'httpd':  
  ensure => running, # Service must be running  
  enable => true,    # Service must be enabled at boot time  
}
```

For inline documentation about a resource, use the `describe` subcommand, for example:

puppet describe file



For a complete reference of the native resource types and their arguments check: <http://docs.puppetlabs.com/references/latest/type.html>

The resource abstraction layer

From the previous resource examples, we can deduce that the Puppet DSL allows us to concentrate on the types of objects (resources) to manage and doesn't bother us on how these resources may be applied on different operating systems.

This is one of Puppet's strong points, resources are **abstracted** from the underlying OS, we don't have to care or specify how, for example, to install a package on Red Hat Linux, Debian, Solaris, or Mac OS, we just have to provide a valid package name. This is possible thanks to Puppet's **Resource Abstraction Layer (RAL)**, which is engineered around the concept of **types** and **providers**.

Types, as we have seen, map to an object on the system. There are more than 50 native types in Puppet (some of them applicable only to specific OSes), the most common and used are `augeas`, `cron`, `exec`, `file`, `group`, `host`, `mount`, `package`, `service`, and `user`. To have a look at their Ruby code, and learn how to make custom types, check these files:

```
ls -l $(factor rubysitedir)/puppet/type
```

For each type, there is at least one provider, which is the component that enables that type on a specific OS. For example, the `package` type is known for having a large number of providers that manage the installation of packages on many OSes, which are `aix`, `appdmg`, `apple`, `aptitude`, `apt`, `aptrpm`, `blastwave`, `dpkg`, `fink`, `freebsd`, `gem`, `hpux`, `macports`, `msi`, `nim`, `openbsd`, `pacman`, `pip`, `pkgdmg`, `pkg`, `pkgutil`, `portage`, `ports`, `rpm`, `rug`, `sunfreeware`, `sun`, `up2date`, `urpmi`, `yum`, and `zypper`.

We can find them here:

```
ls -l $(factor rubysitedir)/puppet/provider/package/
```

The Puppet executable offers a powerful subcommand to interrogate and operate with the RAL: `puppet resource`.

For a list of all the users present on the system, type:

```
puppet resource user
```

For a specific user, type:

```
puppet resource user root
```

Other examples that might give glimpses of the power of RAL to map systems' resources are:

```
puppet resource package
```

```
puppet resource mount
```

```
puppet resource host
```

```
puppet resource file /etc/hosts
```

```
puppet resource service
```

The output is in the Puppet DSL format; we can use it in our manifests to reproduce that resource wherever we want.

The Puppet `resource` subcommand can also be used to modify the properties of a resource directly from the command line, and, since it uses the Puppet RAL, we don't have to know how to do that on a specific OS, for example, to enable the `httpd` service:

```
puppet resource service httpd ensure=running enable=true
```

Nodes

We can place the preceding resources in our first manifest file (`/etc/puppetlabs/code/environments/production/manifests/site.pp`) or in the form included there and they will be applied to all our Puppet managed nodes. This is okay for quick samples out of books, but in real life things are very different. We have hundreds of different resources to manage, and dozens, hundreds, or thousands of different systems to apply different logic and properties to.

To help organize our Puppet code, there are two different language elements: with `node`, we can confine resources to a given host and apply them only to it; with `class`, we can group different resources (or other classes), which generally have a common function or task.

Whatever is declared in a node, definition is included only in the catalog compiled for that node. The general syntax is:

```
node $name [inherits $parent_node] {
  [ Puppet code, resources and classes applied to the node ]
}
```

`$name` is the `certname` of the client (by default its FQDN) or a regular expression; it's possible to inherit, in a node, whatever is defined in the parent node, and, inside the curly braces, we can place any kind of Puppet code: resources declarations, classes inclusions, and variable definitions. An example is given as follows:

```
node 'mysql.example.com' {

  package { ['mysql-server':
    ensure => present,
  ]
  service { ['mysql':
    ensure => 'running',
  ]
}
```

But generally, in nodes we just include classes, so a better real life example would be:

```
node 'mysql.example.com' {
  include common
  include mysql
}
```

The preceding include statements that do what we might expect; they include all the resources declared in the referred class.

Note that there are alternatives to the usage of the node statement; we can use an **External Node Classifier (ENC)** to define which variables and classes assign to nodes or we can have a **nodeless** setup, where resources applied to nodes are defined in a case statement based on the hostname or a similar fact that identifies a node.

Classes and defines

A class can be **defined** (resources provided by the class are defined for later usage but are not yet included in the catalog) with this syntax:

```
class mysql {
  $mysql_service_name = $::osfamily ? {
    'RedHat' => 'mysqld',
    default  => 'mysql',
  }
  package { 'mysql-server':
    ensure => present,
  }
  service { 'mysql':
    name => $mysql_service_name,
    ensure => 'running',
  }
  [...]
}
```

Once defined, a class can be **declared** (the resources provided by the class are actually included in the catalog) in multiple ways:

- Just by including it (we can include the same class many times, but it's evaluated only once):

```
include mysql
```
- By requiring it—what makes all resources in current scope require the included class:

```
require mysql
```

- Containing it—what makes all resources requiring the parent class also require the contained class. In the next example, all resources in `mysql` and in `mysql::service` will be resolved before `exec`:

```
class mysql {
  contain mysql::service
  ...
}

include mysql
exec { 'revoke_default_grants.sh':
  require => Class['mysql'],
}
```

- Using the parameterized style (available since Puppet 2.6), where we can optionally pass parameters to the class, if available (we can declare a class with this syntax only once for each node in our catalog):

```
class { 'mysql':
  root_password => 's3cr3t',
}
```

A parameterized class has a syntax like this:

```
class mysql (
  $root_password,
  $config_file_template = undef,
  ...
) {
  [...]
}
```

Here, we can see the expected parameters defined between parentheses. Parameters with an assigned value have it as their default, as it is here. The case of `undef` for the `$config_file_template` parameter.

The declaration of a parameterized class has exactly the same syntax of a normal resource:

```
class { 'mysql':
  $root_password => 's3cr3t',
}
```

Puppet 3.0 introduced a feature called **data binding**; if we don't pass a value for a given parameter, as in the preceding example, before using the default value, if present, Puppet does an automatic lookup to a Hiera variable with the name `$class::$parameter`. In this example, it would be `mysql::root_password`.

This is an important feature that radically changes the approach of how to manage data in Puppet architectures. We will come back to this topic in the following chapters.

Besides classes, Puppet also has defines, which can be considered classes that can be used multiple times on the same host (with a different title). Defines are also called **defined types**, since they are types that can be defined using Puppet DSL, contrary to the native types written in Ruby.

They have a similar syntax to this:

```
define mysql::user (
    $password,                # Mandatory parameter, no defaults set
    $host = 'localhost',     # Parameter with a default value
    [...]
) {
    # Here all the resources
}
```

They are used in a similar way:

```
mysql::user { 'al':
    password => 'secret',
}
```

Note that defines (also called user **defined types**, **defined resource type**, or **definitions**) like the preceding one, even if written in Puppet DSL, have exactly the same usage pattern as **native types**, written in Ruby (such as package, service, file, and so on).

In types, besides the parameters explicitly exposed, there are two variables that are automatically set. They are `$title` (which is the defined title) and `$name` (which defaults to the value of `$title`) and can be set to an alternative value.

Since a define can be declared more than once inside a catalog (with different titles), it's important to avoid to declare resources with a static title inside a define. For example, this is wrong:

```
define mysql::user ( ...) {
    exec { 'create_mysql_user':
        [ ... ]
    }
}
```

Because, when there are two different `mysql::user` declarations, it will generate an error like:

Duplicate definition: Exec[create_mysql_user] is already defined in file /etc/puppet/modules/mysql/manifests/user.pp at line 2; cannot redefine at /etc/puppet/modules/mysql/manifests/user.pp:2 on node test.example42.com

A correct version could use the `$title` variable which is inherently different each time:

```
define mysql::user ( ...) {
  exec { "create_mysql_user_${title}":
    [ ... ]
  }
}
```

Class inheritance

We have seen that in Puppet classes are just containers of resources that have nothing to do with Object Oriented Programming classes so the meaning of class inheritance is somehow limited to a few specific cases.

When using class inheritance, the parent class (`puppet` in the sample below) is always evaluated first and all the variables and resource defaults sets are available in the scope of the child class (`puppet::server`).

Moreover, the child class can override the arguments of a resource defined in the parent class:

```
class puppet {
  file { ['/etc/puppet/puppet.conf':
    content => template('puppet/client/puppet.conf'),
  ]
}
class puppet::server inherits puppet {
  File['/etc/puppet/puppet.conf'] {
    content => template('puppet/server/puppet.conf'),
  }
}
```

Note the syntax used; when declaring a resource, we use a syntax such as `file { ['/etc/puppet/puppet.conf': [...] ]`; when referring to it the syntax is `File['/etc/puppet/puppet.conf']`.

Even when possible, class inheritance is usually discouraged in Puppet style guides except for some design patterns that we'll see later in the book.

Resource defaults

It is possible to set default argument values for a resource type in order to reduce code duplication. The general syntax to define a resource default is:

```
Type {  
  argument => default_value,  
}
```

Some common examples are:

```
Exec {  
  path => '/sbin:/bin:/usr/sbin:/usr/bin',  
}  
File {  
  mode   => 0644,  
  owner  => 'root',  
  group  => 'root',  
}
```

Resource defaults can be overridden when declaring a specific resource of the same type.

It is worth noting that the *area of effect* of resource defaults might bring unexpected results. The general suggestion is as follows:

- Place the `global` resource defaults in `site.pp` outside any node definition
- Place the `local` resource defaults at the beginning of a class that uses them (mostly for clarity's sake, as they are parse-order independent)

We cannot expect a resource default defined in a class to be working in another class, unless it is a child class, with an inheritance relationship.

Resource references

In Puppet, any resource is uniquely identified by its type and its name. We cannot have two resources of the same type with the same name in a node's catalog.

We have seen that we declare resources with a syntax such as:

```
type { 'name':  
  arguments => values,  
}
```

When we need to reference them (typically when we define dependencies between resources) in our code, the syntax is (note the square brackets and the capital letter):

```
Type['name']
```

Some examples are as follows:

```
file { 'motd': ... }
apache::virtualhost { 'example42.com': .... }
exec { 'download_myapp': .... }
```

These examples are referenced, respectively, with the following code:

```
File['motd']
Apache::Virtualhost['example42.com']
Exec['download_myapp']
```

Variables, facts, and scopes

When writing our manifests, we can set and use variables; they help us in organizing which resources we want to apply, how they are parameterized, and how they change according to our logic, infrastructure, and needs.

They may have different sources:

- **Factor** (variables, called facts, automatically generated on the Puppet client)
- User-defined variables in **Puppet** code (variables defined using Puppet DSL)
- User-defined variables from an **ENC**
- User-defined variables on **Hiera**
- Puppet's **built-in** variables

System's facts

When we install Puppet on a system, the `facter` package is installed as a dependency. Facter is executed on the client each time Puppet is run and it collects a large set of key/value pairs that reflect many system's properties. They are called facts and provide valuable information like the system's `operatingsystem`, `operatingsystemrelease`, `osfamily`, `ipaddress`, `hostname`, `fqdn`, `macaddress` to name just some of the most used ones.

All the facts gathered on the client are available as variables to the Puppet Master and can be used inside manifests to provide a catalog that fits the client.

We can see all the facts of our nodes, running locally:

```
factor -p
```

(The `-p` argument is the short version of `--puppet`, and also shows eventual custom facts, which are added to the native ones, via our modules).

In `factor 1.x`, only plain values were available; `factor 2.x` introduces structured values, so any fact can contain arrays or hashes. `Factor` is replaced by `cFactor` for 3.0, a more efficient implementation in C++ that makes an extensive use of structured data. In any case, it keeps legacy keys, making these two queries equivalent:

```
$ factor ipaddress
1.2.3.4
$ factor networking.interfaces.eth0.ip
1.2.3.4
```

External facts

External facts, supported since Puppet 3.4/`Factor 2.0.1`, provide a way to add facts from arbitrary commands or text files.

These external facts can be added in different ways:

- From modules, by placing them under `facts.d` inside the module root directory
- In directories within nodes:
 - In a directory specified by the `--external-dir` option
 - In the Linux and Mac OS X in `/etc/puppetlabs/facter/facts.d/` or `/etc/facts.d/`
 - In Windows in `C:\ProgramData\PuppetLabs\facter\facts.d\`
 - When running as a non-root user in `$HOME/.facter/facts.d/`

Executable facts can be scripts in any language, or even binaries; the only requirement is that its output has to be formed by lines with the format `key=value`, like:

```
key1=value1
key2=value2
```

Structured data facts have to be plain text files with an extension indicating its format, `.txt` files for files containing `key=value` lines, `.yaml` for YAML files, and `.json` for JSON files.

User variables in Puppet DSL

Variable definition inside the Puppet DSL follows the general syntax: `$variable = value`.

Let's see some examples. Here the value is set as a string, a boolean, an array, or a hash:

```
$redis_package_name = 'redis'
$install_java = true
$dns_servers = [ '8.8.8.8' , '8.8.4.4' ]
$config_hash = { user => 'joe', group => 'admin' }
```

From Puppet 3.5, using the future parser or starting on version 4.0 by default Here docs are also supported, what is a convenient way of define multiline strings:

```
$gitconfig = $("GITCONFIG")
[user]
  name = ${git_name}
  email = ${email}
| GITCONFIG
file { "${homedir}/.gitconfig":
  content => $gitconfig,
}
```

They have multiple options. In the previous example, we set `GITCONFIG` as the delimiter, the quotes indicate that variables in the text have to be interpolated, and the pipe character marks the indentation level.

Here, the value is the result of a **function** call (which may have strings and other data types or other variables as arguments):

```
$config_file_content = template('motd/motd.erb')

$dns_servers = hiera(name_servers)
$dns_servers_count = inline_template('<%= @dns_servers.length %>')
```

Here, the value is determined according to the value of another variable (here the `$:osfamily` fact is used), using the `selector` construct:

```
$mysql_service_name = $::osfamily ? {
  'RedHat' => 'mysqld',
  default => 'mysql',
}
```

A special value for a variable is `undef` (a null value similar to Ruby's `nil`), which basically removes any value to the variable (can be useful in resources when we want to disable, and make Puppet ignore, an existing attribute):

```
$config_file_source = undef
file { '/etc/motd':
  source => $config_file_source,
  content => $config_file_content,
}
```

Note that we can't change the value assigned to a variable inside the same class (more precisely inside the same scope; we will review them later). Consider a code like the following:

```
$counter = '1'
$counter = $counter + 1
```

The preceding code will produce the following error:

```
Cannot reassign variable counter
```

Type-checking

The new parser used as default in Puppet 4 has better support for data types, what includes optional type-checking. Each value in Puppet has a data type, for example, strings are of the type `String`, booleans are of the type `Boolean`, and types themselves have their own type `Type`. Every time we declare a parameter, we can enforce its type:

```
class ntp (
  Boolean $enable = true,
  Array[String] $servers = [],
) { ... }
```

We can also check types with expressions like this one:

```
$is_boolean =~ String
```

Or make selections by the type:

```
$enable_real = $enable ? {
  Boolean => $enable,
  String  => str2bool($enable),
  Numeric => num2bool($enable),
  default => fail('Illegal value for $enable parameter')
}
```

Types can have parameters, `String[8]` would be a string of at least 8 characters-length, `Array[String]` would be an array of strings and `Variant[Boolean, Enum['true', 'false']]` would be a composed value that would match with any `Boolean` or with members of the enumeration formed by the strings `true` and `false`.

User variables in an ENC

When an ENC is used for the classification of nodes, it returns the classes to include in the requested node and variables. All the variables provided by an ENC are at top scope (we can reference them with `$.variablename` all over our manifests).

User variables in Hiera

A very popular and useful place to place user data (yes, variables) is also Hiera; we will review it extensively in *Chapter 2, Managing Puppet Data with Hiera*; let's just point out a few basic usage patterns here. We can use it to manage any kind of variable, whose value can change according to custom logic in a hierarchical way. Inside manifests, we can lookup a Hiera variable using the `hiera()` function. Some examples are as follows:

```
$dns = hiera(dnsservers)
class { 'resolver':
  dns_server => $dns,
}
```

The preceding example can also be written as:

```
class { 'resolver':
  dns_server => hiera(dnsservers),
}
```

In our Hiera YAML files, we would have something like this:

```
dnsservers:
- 8.8.8.8
- 8.8.4.4
```

If our Puppet Master uses Puppet version 3 or greater, then we can benefit from the Hiera automatic lookup for class parameters, that is, the ability to define values for any parameter exposed by the class in Hiera. The preceding example would become something like this:

```
include resolver
```

Then, in the Hiera YAML files:

```
resolver::dns_server:  
  - 8.8.8.8  
  - 8.8.4.4
```

Puppet built-in variables

A bunch of other variables are available and can be used in manifests or templates:

- **Variables set by the client (agent):**
 - `$clientcert`: This is the name of the node (`certname` setting in its `puppet.conf`, by default is the host's FQDN)
 - `$clientversion`: This is the Puppet version on the agent
- **Variables set by the server (Master):**
 - `$environment`: This is a very important special variable, which defines the Puppet's environment of a node (for different environments the Puppet Master can serve manifests and modules from different paths)
 - `$servername`, `$serverip`: These are respectively, the Master's FQDN and IP address
 - `$serverversion`: This is the Puppet version on the Master (is always better to have Masters with Puppet version equal or newer than the clients)
 - `$settings::<setting_name>`: This is any configuration setting of the Puppet Master's `puppet.conf` variable
- **Variables set by the parser during catalog compilation:**
 - `$module_name`: This is the name of the module that contains the current resource definition
 - `$caller_module_name`: This is the name of the module that contains the current resource declaration

Variables scope

One of the parts where Puppet development can be misleading and not so intuitive is how variables are evaluated according to the place in the code where they are used.

Variables have to be declared before they can be used and this is parse order dependent, so, for this reason, Puppet language can't be considered completely declarative.

In Puppet, there are different scopes; partially isolated areas of code where variables and resource defaults values can be confined and accessed.

There are four types of scope, from general to local:

- **Top scope:** This is any code defined outside nodes and classes, as what is generally placed in `/etc/puppet/manifests/site.pp`
- **Node scope:** This is code defined inside nodes definitions
- **Class scope:** This is code defined inside a class or define
- **Sub class scope:** This is code defined in a class that inherits another class

We always write code within a scope and we can directly access variables (that is just specifying their name without using the fully qualified name) defined only in the same scope or in a parent or containing one. The following are the ways, we can access top scope variables, node scope variables, and class variables:

- Top scope variables can be accessed from anywhere
- Node scope variables can be accessed in classes (used by the node), but not at the top scope
- Class (also called local) variables are directly available, with their plain name, only from within the same class or define where they are set or in a child class

Variables' value or resources default arguments defined at a more general level can be overridden at a local level (Puppet uses always the most local value).

It's possible to refer to variables outside a scope by specifying their fully qualified name, which contains the name of the class where the variables are defined. For example, `$::apache::config_dir` is a variable, called `config_dir`, defined in the `apache` class.

One important change introduced in Puppet 3.x is the forcing of **static scoping** for variables; this involves that a parent scope for a class can be only its parent class.

Earlier Puppet versions had **dynamic scoping**, where parent scopes were assigned both by inheritance (as in static scoping) and by simple declaration; that is, any class has the *first* scope where it has been declared as parent. This means that, since we can include classes multiple times, the order used by Puppet to parse our manifests may change the parent scope and therefore how a variable is evaluated.

This can obviously lead to any kind of unexpected problems, if we are not particularly careful on how classes are declared, with variables evaluated in different (parse order dependent) ways. The solution is Puppet 3's static scoping and the need to reference to out of scope variables with their fully qualified name.

Meta parameters

Meta parameters are general-purpose parameters available to any resource type, even if not explicitly defined. They can be used for different purposes:

- **Manage dependencies and resources ordering (more on them in the next section):** `before`, `require`, `subscribe`, `notify`, and `stage`
- **Manage resources' application policies:** `audit` (audit the changes done on the attributes of a resource), `noop` (do not apply any real change for a resource), `schedule` (apply the resources only within a given time schedule), and `loglevel` (manage the log verbosity)
- **Add information to a resource:** `alias` (adds an alias that can be used to reference a resource) and `tag` (add a tag that can be used to refer to group resources according to custom needs; we will see a usage case later in this chapter in the external resources section)

Managing order and dependencies

The Puppet language is declarative and not procedural; it defines **states** as follows: the order in which resources are written in manifests does not affect the order in which they are applied to the desired state.



The Puppet language is declarative and not procedural. This is not entirely true – contrary to resources, variables definitions are parse order dependent, so the order used to define variables is important. As a general rule, just set variables before using them, which sounds logical, but is procedural.

There are cases where we need to set some kind of ordering among resources, for example, we might want to manage a configuration file only after the relevant package has been installed, or have a service automatically restart when its configuration files change. Also, we may want to install packages only after we've configured our packaging systems (apt sources, yum repos, and so on) or install our application only after the whole system and the middleware has been configured.

To manage these cases, there are three different methods, which can coexist:

- Use the **meta parameters** `before`, `require`, `notify`, and `subscribe`
- Use the **chaining arrows operator** (respective to the preceding meta parameters: `->`, `<-`, `<~`, `~>`)
- Use **run stages**

In a typical package/service/configuration file example, we want the package to be installed first, configure it, and then start the service, eventually managing its restart if the config file changes.

This can be expressed with meta parameters:

```
package { 'exim':  
  before => File['exim.conf'],  
}  
file { 'exim.conf':  
  notify => Service['exim'],  
}  
service { 'exim': }
```

This is equivalent to this chaining arrows syntax:

```
package {'exim': } ->  
file {'exim.conf': } ~>  
service{'exim': }
```

However, the same ordering can be expressed using the alternative *reverse* meta parameters:

```
package { 'exim': }  
file { 'exim.conf':  
  require => Package['exim'],  
}  
service { 'exim':  
  subscribe => File['exim.conf'],  
}
```

They can also be expressed like this:

```
service{'exim': } <~  
file{'exim.conf': } <-  
package{'exim': }
```

Run stages

Puppet 2.6 introduced the concept of **run stages** to help users manage the order of dependencies when applying groups of resources.

Puppet provides a default main stage; we can add any number or further stages, and their ordering, with the `stage` **resource type** and the normal syntax we have seen:

```
stage { 'pre':  
  before => Stage['main'],  
}
```

The normal syntax is equivalent to:

```
stage { 'pre': }  
Stage['pre'] -> Stage['main']
```

We can assign any class to a defined stage with the `stage meta` parameter:

```
class { 'yum':  
  stage => 'pre',  
}
```

In this way, all the resources provided by the `yum` class are applied before all the other resources (in the default main stage).

The idea of stages at the beginning seemed a good solution to better handle large sets of dependencies in Puppet. In reality, some drawbacks and the augmented risk of having dependency cycles make them less useful than expected. A thumb rule is to use them for simple classes (that don't include other classes) and where it is really necessary (for example, to set up package management configurations at the beginning of a Puppet run or deploy our application after all the other resources have been managed).

Reserved names and allowed characters

As with every language, Puppet DSL has some restrictions on the names we can give to its elements and the allowed characters.

As a general rule, for names of resources, variables, parameters, classes, and modules we can use only **lowercase letters**, **numbers**, and the **underscore** (`_`). Usage of hyphens (`-`) should be avoided (in some cases it is forbidden; in others it depends on Puppet's version).

We can use uppercase letters in variables names (but not at their beginning) and any character for resources' titles.

Names are case sensitive, and there are some reserved words that cannot be used as names for resources, classes, or defines or as unquoted word strings in the code, such as:

`and, case, class, default, define, else, elsif, false, if, in, import, inherits, node, or, true, undef, unless, main, settings, $string.`

A fully updated list of reserved words can be found here: https://docs.puppet.com/puppet/latest/reference/lang_reserved.html

Conditionals

Puppet provides different constructs to manage conditionals inside manifests.

Selectors as we have seen, let us set the value of a variable or an argument inside a resource declaration according to the value of another variable. Selectors, therefore, just return values, and are not used to manage conditionally entire blocks of code.

Here's an example of a selector:

```
$package_name = $::osfamily ? {  
  'RedHat' => 'httpd',  
  'Debian' => 'apache2',  
  default  => undef,  
}
```

The `case` statements are used to execute different blocks of code according to the values of a variable. It's recommended to have a default block for unmatched entries. Case statements can't be used inside resource declarations. We can achieve the same result of the previous selector with this `case` sample:

```
case $::osfamily {  
  'Debian': { $package_name = 'apache2' }  
}
```

```
'RedHat': { $package_name = 'httpd' }
default: { fail ("Operating system $::operatingsystem not
supported") }
}
```

The `if`, `elsif`, and `else` conditionals, like `case`, are used to execute different blocks of code and can't be used inside resource declarations. We can use any of Puppet's comparison expressions and we can combine more than one for complex patterns matching.

The previous sample variables assignment can also be expressed in this way:

```
if $::osfamily == 'Debian' {
  $package_name = 'apache2'
} elsif $::osfamily == 'RedHat' {
  $package_name = 'httpd'
} else {
  fail ("Operating system $::operatingsystem not supported")
}
```

The `unless` condition is the opposite of `if`. It evaluates a Boolean condition and, if it's false, it executes a block of code. The use of `unless` instead of negating the `if` condition is more a personal preference, but it shouldn't be used with complex expressions as it reduces readability.

Comparison operators

Puppet supports some common comparison operators, which resolve to true or false:

- Equal `==`, returns true if the operands are equal. Used with numbers, strings, arrays, hashes, and Booleans. For example:

```
if $::osfamily == 'Debian' { [ ... ] }
```
- Not equal `!=`, returns true if the operands are different:

```
if $::kernel != 'Linux' { [ ... ] }
```
- Less than `<`, greater than `>`, less than or equal to `<=`, and greater than or equal to `>=` can be used to compare numbers:

```
if $::uptime_days > 365 { [ ... ] }
if $::operatingsystemrelease <= 6 { [ ... ] }
```

- Regex match `=~` compares a string (left operator) with a regular expression (right operator), and resolves true, if it matches. Regular expressions are enclosed between forward slashes and follow the normal Ruby syntax:

```
if $mode =~ /(server|client)/ { [ ... ] }
if $::ipaddress =~ /^10\./ { [ ... ] }
```
- Regex not match `!~`, opposite to `=~`, resolves false if the operands match.

Iteration and lambdas

Puppet language has historically been somehow limited in iterators, it didn't have explicit support for this till version 4.0. The old way of doing it is by the use of defined types. All Puppet resources can have an array as its title, which is equivalent to creating the same resource one time with each of the elements of the array.

This approach, although sometimes convenient and orthogonal with the rest of the language, has some limitations. First, only the title varies between each created resource, which limits the possibilities of the code in the iteration, and second, a defined type needs to be implemented just for the iteration; it can even happen that the type is defined far from the place where we want to iterate, thus over-complicating it and making it less readable. Here is an example:

```
define nginx::enable_site ($site = $title) {
  file { ["/etc/nginx/sites-enabled/$site":
    ensure => link,
    target => "/etc/nginx/sites-available/$site",
  ]
}
$sites = ['example.com', 'test.puppetlabs.com']
nginx::enable_site { $sites }
```

In newer versions, the language includes support for lambda functions and some functions that accept these lambdas as parameters, allowing more explicit iterators, for example, to define resources:

```
$sites = ['example.com', 'test.puppetlabs.com']
$sites.each |String $value| {
  file { ["/etc/nginx/sites-enabled/$site":
    ensure => link,
    target => "/etc/nginx/sites-available/$site",
  ]
}
```

To transform data, like selecting the sites that start with "test" from a list, use a code as follows:

```
$test_sites = $sites.filter |$site| { $site =~ /^test\. / }
```

The in operator

The `in` operator checks whether a string is present in another string, an array, or in the keys of a hash. It is case sensitive:

```
if '64' in $::architecture
if $monitor_tool in [ 'nagios' , 'icinga' , 'sensu' ]
```

Expressions combinations

It's possible to combine multiple comparisons with `and` and `or`:

```
if ($::osfamily == 'RedHat') and ($::operatingsystemrelease == '5') {
[ ... ] }
if (operatingsystem == 'Ubuntu') or ($::operatingsystem == 'Mint') { [
... ] }
```

Exported resources

When we need to provide information to a host about resources present in another host, things in Puppet become trickier. This can be needed for example, for monitoring or backup solutions. The only official solution has been, for a long time, to use **exported resources**; resources declared in the catalog of a node (based on its facts and variables), but applied (collected) on another node. Some alternative approaches are now possible with PuppetDB, we will review them in *Chapter 3, Introducing PuppetDB*.

Resources are declared with the special `@@` notation, which marks them as exported so that they are not applied to the node where they are declared:

```
@@host { $::fqdn:
  ip => $::ipaddress,
}
@@concat::fragment { "balance-fe-${::hostname}":
  target => '/etc/haproxy/haproxy.cfg',
  content => "server ${::hostname} ${::ipaddress} maxconn 5000",
  tag    => "balance-fe",
}
```

Once a catalog containing exported resources has been applied on a node and stored by the Puppet Master, the exported resources can be collected with the `<<| |>>` operator, where it is possible to specify search queries:

```
Host <<| |>>
Concat::Fragment <<| tag == "balance-fe" |>>
Sshkey <<| |>>
Nagios_service <<| |>>
```

In order to use exported resources, we need to enable on the Puppet Master the `storeconfigs` option and specify the backend to be used. For a long time, the only available backend was Rails' active records, which typically used MySQL for data persistence. This solution was the best for its time, but suffered severe scaling limitations. Luckily, things have changed, a lot, with the introduction of PuppetDB, which is a fast and reliable storage solution for all the data generated by Puppet, including exported resources.

Virtual resources

Virtual resources define a desired state for a resource without adding it to the catalog. Like normal resources, they are applied only on the node where they are declared, but, as virtual resources, we can apply only a subset of the ones we have declared; they have also a similar usage syntax: we declare them with a single `@` prefix (instead of the `@@` used for exported resources), and we collect them with `<| |>` (instead of `<<| |>>`).

A useful and rather typical example involves user's management.

We can declare all our users in a single class, included by all our nodes:

```
class my_users {
  @user { 'al': [...] tag => 'admins' }
  @user { 'matt': [...] tag => 'developers' }
  @user { 'joe': [...] tag => 'admins' }
  [ ... ]
}
```

These users are actually not created on the system; we can decide which ones we actually want on a specific node with a syntax like this:

```
User <| tag == admins |>It is equivalent to:
realize(User['al'] , User['joe'])
```

Note that the `realize` function needs to address resources with their name.

Modules

Modules are self-contained, distributable, and (ideally) reusable recipes to manage specific applications or system's elements.

They are basically just a directory with a predefined and standard structure that enforces configuration over naming conventions for the managed provided classes, extensions, and files.

The `$modulepath` configuration entry defines where modules are searched; this can be a list of colon separated directories.

Paths of a module and auto loading

Modules have a standard structure, for example, for a MySQL module the code reads thus:

```
mysql/                # Main module directory

mysql/manifests/      # Manifests directory. Puppet code here.
mysql/lib/             # Plugins directory. Ruby code here
mysql/templates/      # ERB Templates directory
mysql/files/           # Static files directory
mysql/spec/           # Puppet-rspec test directory
mysql/tests/          # Tests / Usage examples directory
mysql/facts.d/         # Directory for external facts

mysql/Modulefile       # Module's metadata descriptor
```

This layout enables useful conventions, which are widely used in Puppet world; we must know them to understand where to look for files and classes:

For example, we can use modules and write the following code:

```
include mysql
```

Puppet will then automatically look for a class called `mysql` defined in the file `$modulepath/mysql/manifests/init.pp`:

The `init.pp` script is a special case that applies for classes that have the same name of the module. For sub classes there's a similar convention that takes in consideration the subclass name:

```
include mysql::server
```

It then auto loads the `$modulepath/mysql/manifests/server.pp` file.

A similar scheme is also followed for defines or classes at lower levels:

```
mysql::conf { ...}
```

This define is searched in `$modulepath/mysql/manifests/conf.pp`:

```
include mysql::server::ha
```

It then looks for `$modulepath/mysql/manifests/server/ha.pp`.

It's generally recommended to follow these naming conventions that allow auto loading of classes and defines without the need to explicitly import the manifests that contain them.



Note that, even if not considered good practice, we can currently define more than one class or define inside the same manifest as, when Puppet parses a manifest, it parses its whole contents.

Module's naming conventions apply also to the files that Puppet provides to clients.

We have seen that the `file` resource accepts two different and alternative arguments to manage the content of a file: `source` and `content`. Both of them have a naming convention when used inside modules.

Templates, typically parsed via the `template` or the `epp` functions with syntax like the one given here, are found in a place like `$modulepath/mysql/templates/my.cnf.erb`:

```
content => template('mysql/my.cnf.erb'),
```

This also applies to sub directories, so for example:

```
content => template('apache/vhost/vhost.conf.erb'),
```

It uses a template located in `$modulepath/apache/templates/vhost/vhost.conf.erb`.

A similar approach is followed with static files provided via the `source` argument:

```
source => 'puppet:///modules/mysql/my.cnf'
```

It serves a file placed in `$modulepath/mysql/files/my.cnf`:

```
source => 'puppet:///modules/site/openssh/sshd_config'
```

This serves a file placed in `$modulepath/site/openssh/sshd_config`.

Notice the differences in templates and source paths. Templates are resolved in the server, and they are always placed inside the modules. Sources are retrieved by the client and modules in the URL could be a different mount point if it's configured on the server.

Finally, the whole content of the `lib` subdirectory in a module has a standard scheme. Note that here, we can place Ruby code that extends Puppet's functionality and is automatically redistributed from the Master to all clients (if the `pluginsync` configuration parameter is set to `true`, this is default for Puppet 3 and widely recommended in any setup):

```
mysql/lib/augeas/lenses/          # Custom Augeas lenses.
mysql/lib/facter/                 # Custom facts.
mysql/lib/puppet/type/            # Custom types.
mysql/lib/puppet/provider/<type_name>/ # Custom providers.
mysql/lib/puppet/parser/functions/ # Custom functions.
```

Templates

Files provisioned by Puppet can be templates written in Ruby's ERB templating language or in the Embedded Puppet Template Syntax (EPP).

An ERB template can contain whatever text we need and have inside `<% %>` tags, interpolation of variables or Ruby code. We can access, in a template, all the Puppet variables (facts or user assigned) with the `<%= tag`:

```
# File managed by Puppet on <%= @fqdn %>
search <%= @domain %>
```

The `@` prefix for variable names is highly recommended in all Puppet versions, and mandatory starting from 4.0.

To use out of scope variables, we can use the `scope.lookupvar` method:

```
path <%= scope.lookupvar('apache::vhost_dir') %>
```

This uses the variable's fully qualified name. If the variable is at top scope then run the following command:

```
path <%= scope.lookupvar('::fqdn') %>
```

Since Puppet 3, we can use this alternative syntax:

```
path <%= scope['apache::vhost_dir'] %>
```


In ERB templates, we can also use more elaborate Ruby code inside a `<%` opening tag, for example, to reiterate over an array:

```
<% @dns_servers.each do |ns| %>
  nameserver <%= ns %>
<% end %>
```

The `<%` tag is used to place a line of text if some conditions are met:

```
<% if scope.lookupvar('puppet::db') == "puppetdb" -%>
  storeconfigs_backend = puppetdb
<% end -%>
```

Noticed the `-%>` ending tag here? When the dash is present, no line is introduced on the generated file, as it would if we had written `<% end %>`.

EPP templates are quite similar, they are also plain text files and they use the same tags for the embedded code, the main differences are that EPPs use Puppet code instead of Ruby, that they can receive type-checked parameters, and that they can directly access other variables where in ERBs we'd have to use lookup functions.

The parameters definition is optional but if it's included it has to be in the beginning of the file:

```
<%- | Array[String] $dns_servers,
      String $search_domain | -%>
```

To use templates in Puppet code, we have to use the `template` function for ERBs or the `epp` function for EPPs; `epp` can receive a hash with the values of the arguments as a second argument:

```
file { '/etc/resolv.conf':
  content => epp('resolvconf/resolv.conf.epp', {
    'dns_servers': ['8.8.8.8', '8.8.4.4'],
    'search_domain': 'example.com',
  })
}
```

Restoring files from a filebucket

Puppet, by default, makes a local copy of all the files that it changes on a system; it allows the recover old versions of files overwritten by Puppet. This functionality is managed with the `filebucket` type, which allows to store a copy of the original files, either on a central server or locally on the managed system.

When we run Puppet, we see messages like this:

```
info: /Stage[main]/Ntp/File[ntp.conf]: Filebucketed /etc/ntp.conf to
puppet with sum 7fda24f62b1c7ae951db0f746dc6e0cc
```

The checksum of the original file is useful to retrieve it; in fact files are saved in the directory `/var/lib/puppet/clientbucket` in a series of subdirectories named according to the same checksum. So, given the preceding example, our file contents are saved in:

```
/var/lib/puppet/clientbucket/7/f/d/a/2/4/f/6/
7fda24f62b1c7ae951db0f746dc6e0cc/contents
```

We can verify the original path in:

```
/var/lib/puppet/clientbucket/7/f/d/a/2/4/f/6/
7fda24f62b1c7ae951db0f746dc6e0cc/paths
```

A quick way to look for the saved copies of a file, therefore, is to use a command like this:

```
grep -R /etc/ntp.conf /var/lib/puppet/clientbucket/
```

Puppet provides the `filebucket` subcommand to retrieve saved files. In the preceding example, we can recover the original file with a (not particularly handy) command like:

```
puppet filebucket restore -l --bucket /var/lib/puppet/clientbucket /etc/
ntp.conf 7fda24f62b1c7ae951db0f746dc6e0cc
```

It's possible to configure remote `filebucket`, typically on the Puppet Master using the special `filebucket` type:

```
filebucket { 'central':
  path  => false,      # This is required for remote filebuckets.
  server => 'my.s.com', # Optional, by default is the puppetmaster
}
```

Once declared `filebucket`, we can assign it to a file with the `backup` argument:

```
file { '/etc/ntp.conf':
  backup => 'central',
}
```

This is generally done using a resource default defined at top scope (typically in our `/etc/puppet/manifests/site.pp`):

```
File { backup => 'central', }
```

Summary

In this chapter, we have reviewed and summarized the basic Puppet principles that are a prerequisite to better understanding the contents of the book. We have seen how Puppet is configured and what its main components are: manifests, resources, nodes, classes, and the power of the Resource Abstraction Layer.

The most useful language elements have been described as variables, references, resources defaults and ordering, conditionals, iterators, and comparison operators. We took a look at export and virtual resources and analyzed the structure of a module and learned how to work with ERB and EPP templates. Finally, we saw how Puppet's filebucket works and how to recover files modified by Puppet.

We are ready to face a very important component of the Puppet ecosystem: Hieradata and how it can be used to separate our data from Puppet code.

2

Managing Puppet Data with Hiera

The history of Puppet is an interesting example of how best practices have evolved with time, following new usage patterns and contributions from the community.

Once people started to write manifests with Puppet's DSL and express the desired state of their systems, they found themselves placing custom variables and parameters that expressed various resources of their infrastructures (IP addresses, hostnames, paths, URLs, names, properties, lists of objects, and so on) inside the code used to create the needed resource types.

At times, variables were used to classify and categorize nodes (systems' roles, operational environments, and so on), other times facts (such as `$::operatingsystem`) were used to provide resources with the right names and paths according to the underlying OS.

Variables could be defined in different places; they could be set via an **External Node Classifier (ENC)**, inside node declarations or inside classes.

There was not (and actually there isn't) any strict rule on how and where users data could be placed, but the general outcome was that we found ourselves having our custom data defined inside our manifests.

Now, in my very personal and definitely non-orthodox opinion, this is not necessarily or inherently a bad thing; looking at the data we provide when we define our resources gives us clearer visibility on how things are done and doesn't compel us to look in different places to understand what our code is doing.

Nevertheless, such an approach may fit relatively simple setups where we don't need to cope with large chunks of data, which might come from different sources and change a lot according to different factors.

Also, we might need to have different people working on Puppet—who write the code and design its logic and those who need to apply configurations, mostly dealing with data.

More generally, the concept of separating data from code is a well-established and sane development practice that also makes sense in the Puppet world.

The person who faced this issue in the most resolute way is R.I.Pienaar. First, he developed the `extlookup` function (included in Puppet core for a long time), which allows to read data from external CSV files, then he took a further step—developing **Hiera**, a key-value lookup tool where data used by our manifests can be placed and evaluated differently according to a custom hierarchy from different data sources.

One of the greatest features of Hiera is its modular pluggable design that allows the usage of different backends that may retrieve data from different sources: YAML or JSON files, Puppet classes, MySQL, Redis, REST services, and more.

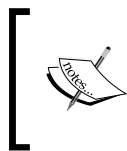
In this chapter, we will cover the following topics:

- Installing and configuring Hiera
- Defining custom hierarchies and backends
- Using the `hiera` command-line tool
- Using the `hiera()`, `hiera_array()`, and `hiera_hash()` functions inside our Puppet manifests
- Integrating Hiera in Puppet 3
- Providing files via Hiera with the `hiera-file` backend
- Encrypting our data with the `hiera-gpg` and `hiera-eyaml` backends
- Using Hiera as an ENC with `hiera_include()` function

Installing and configuring Hiera

From Puppet 3.x, Hiera has been officially integrated, and it is installed as a dependency when we install Puppet.

With Puppet 2.x, we need to install it separately, on the node where the Puppet Master resides—we need both the `hiera` and `hiera-puppet` packages, either via the OS native packaging system or via `gem`.



gem is a package manager for Ruby, the language used to implement Puppet. It offers a unified format for self-contained packages commonly called gems. It's commonly used to install Puppet plugins. We'll see it multiple times throughout the book.

Hiera is not needed by the clients, unless they operate in a Masterless setup as Hiera is only used in the variables lookup during catalog compilation.

Its configuration file is `hiera.yaml`, its paths depends on how it is invoked:

- When invoked from Puppet, the default path will be `/etc/puppetlabs/code/hiera.yaml` (`/etc/puppet/hiera.yaml` and `/etc/puppetlabs/puppet/hiera.yaml` for Puppet Enterprise); this can be modified with the `hiera_config` setting in the master section of the `puppet.conf` file
- When invoked from the CLI or when used within the Ruby code, the path is `/etc/hiera.yaml`

When invoked from CLI, we can also specify a configuration file with the `--config` flag: `hiera --config /etc/puppetlabs/code/hiera.yaml`; if `hiera` in this host is only used for Puppet, we can link this config file to the default path `/etc/hiera.yaml` so we don't need to pass the flag to the `hiera` command.

The hiera.yaml configuration file

The file is a YAML hash, where the top-level keys are Ruby symbols, with a colon (`:`) prefix, which may be either global or backend specific settings.

The default content for the configuration file is as follows:

```
---
:backends: yaml
:yaml:
  :datadir: /etc/puppetlabs/code/environments/{environment}/hieradata
:hierarchy:
  - "nodes/{::trusted.certname}"
  - common
:logger: console
```

Using these settings, Hiera key-values are read from a YAML file with the `/etc/puppetlabs/code/environments/{environment}/common.yaml` path.

The default `datadir` in versions before 4.0 was `/var/lib/hiera`.

Global settings

Global settings are general configurations that are independent from the used backend. They are listed as follows:

- `:hierarchy`: This is a string or an array describing the data sources to look for. Data sources are checked from top to bottom and may be dynamic, that is, contain variables (we reference them with `%{variablename}`). The default value is `common`.
- `:backends`: This is a string or an array that defines the backends to be used. The default value is `yaml`.
- `:logger`: This is a string of a logger where messages are sent. The default value is `console`.
- `:merge_behavior`: This is a string that describes how hash values are merged across different data sources. The default value is `native`; the first key found in the hierarchy is returned. Alternative values `deep` and `deeper` require the `deep_merge` Ruby gem.

Backend specific settings

Any backend may have its specific settings; here is what is used by the native YAML, JSON and Puppet backend:

- `:datadir`: This is a string. It is used by the JSON and YAML backends, and it is the directory where the data sources that are defined in the hierarchy can be found. We can place variables (`%{variablename}`) here for a dynamic lookup.
- `:datasource`: This is a string. It is used by the Puppet backend. This is the name of the Puppet class where we have to look for variables.

Examples

A real world configuration that uses the extra GPG backend, used to store encrypted secrets as data, may look like the following:

```
---
:backends:
  - yaml
  - gpg

:hierarchy:
  - "nodes/%{::fqdn}"
  - "roles/%{::role}"
```

```

- "zones/%{::zone}"
- "common"

:yaml:
  :datadir: /etc/puppetlabs/code/environments/%{environment}/hieradata
:pgp:
  :datadir: /etc/puppetlabs/code/environments/%{environment}/hieradata
  :key_dir: /etc/puppetlabs/pgpkeys

```



Note that the preceding example uses custom `::$role` and `::$zone` variables that identify the function of the node and its datacenter, zone, or location. They are not native facts so we should define them as custom facts or as top scope variables.

Note also that an example such as this expects to have modules that fully manage operating systems differences, as recommended, so that we don't have to manage different settings for different OSes in our hierarchy.

Be aware that in the hierarchy array, if individual values begin with a variable to interpolate, we need to use double quotes (").

The following is an example with the usage of the file backend to manage not only key-value entries, but also whole files:

```

---
:backends:
  - yaml
  - file
  - pgp

:hierarchy:
  - "%{::env}/fqdn/%{::fqdn}"
  - "%{::env}/role/%{::role}"
  - "%{::env}/zone/%{::zone}"
  - "%{::env}/common"

:yaml:
  :datadir: /etc/puppetlabs/code/data

:file:
  :datadir: /etc/puppetlabs/code/data

:pgp:
  :key_dir: /etc/puppetlabs/code/pgpkeys
  :datadir: /etc/puppetlabs/code/pgpdata

```




Note that, besides the added backend with its configuration, an alternative approach is used to manage different environments (intended as the operational environments of the nodes, for example production, staging, test, and development).

Here, to identify the node's environment, we use a custom top scope variable or fact called `$::env` and not Puppet's internal variable `$::environment` (to which we can map different module paths and manifest files).

Working with the command line on a YAML backend

When we use a backend based on files such as JSON or YAML, which are the most commonly used, we have to recreate on the filesystem the hierarchy defined in our `hieradata.yaml` file, the files that contain Hieradata data must be placed in these directories.

Let's see Hieradata in action. Look at the following sample hierarchy configuration:

```
:hierarchy:
  - "nodes/%{::fqdn}"
  - "env/%{::env}"
  - common

:yaml:
  :datadir: /etc/puppetlabs/code/hieradata
```

We have to create a directory structure as follows:

```
mkdir -p /etc/puppetlabs/code/hieradata/{nodes,env}
```

Then, work on the YAML files as shown:

```
vi /etc/puppetlabs/code/hieradata/nodes/web01.example42.com.yaml
vi /etc/puppetlabs/code/hieradata/env/production.yaml
vi /etc/puppetlabs/code/hieradata/env/test.yaml
vi /etc/puppetlabs/code/hieradata/common.yaml
```

These files are plain YAML files where we can specify the values for any Hieradata-managed key. These values can be strings, arrays, or hashes:

```
vi /etc/puppet/hieradata/common.yaml

---
# A simple string assigned to a key
```

```

timezone: 'Europe/Rome'

# A string with variable interpolation
nagios_server: "nagios.%{::domain}"

# A string with another variable defined in Hiera (!)
dns_nameservers: "%{hiera('dns_servers')}}"

# A string assigned to a key that maps to the
# template parameter of the openssh class (from Puppet 3)
openssh::template: 'site/common/openssh/sshd_config.erb'

# An array of values
ldap_servers:
  - 10.42.10.31
  - 10.42.10.32

# An array with a single value
ntp::ntp_servers:
  - 10.42.10.71

# A hash of values
users:
  al:
    home: '/home/al'
    comment: 'Al'
  jenkins:
    password: '!'
    comment: 'Jenkins'

```

Given the previous example, execute a Hiera invocation as follows:

```
hiera ldap_servers
```

It will return the following array:

```
["10.42.10.31", "10.42.10.32"]
```

If we define a different value for a key in a data source that is higher in the hierarchy, then that value is returned. Let's create a data source for the `test` environment as follows:

```
vi /etc/puppet/hieradata/env/test.yaml
```

```

---
ldap_servers:

```

```
- 192.168.0.31
users:
  qa:
    home: '/home/qa'
    comment: 'QA Tester'
```

A normal Hieradata lookup for the `ldap_servers` key will still return the common value, as follows:

```
hieradata ldap_servers
["10.42.10.31", "10.42.10.32"]
```

If we explicitly pass the `env` variable, the returned value is the one for the `env` test as follows:

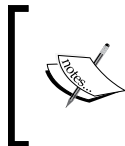
```
hieradata ldap_servers env=test
["192.168.0.31"]
```

If we have a more specific setting for a given node, that value is returned:

```
vi /etc/puppet/hieradata/nodes/ldap.example42.com.yaml
```

```
---
ldap_servers:
- 127.0.0.1
```

```
hieradata ldap_servers fqdn=ldap.example42.com
["127.0.0.1"]
```



We have seen that, by default, Hieradata returns the first value found while traversing the data sources hierarchy, from the first to the last. When more backends are specified, the whole hierarchy of the first backend is fully traversed, then the same is done for the second and so on.

Hieradata also provides some alternative lookup options. When we deal with arrays, for example, we can decide to merge all the values found in the hierarchy, instead of returning the first one found. Let's see how the result of the preceding command changes if the array lookup is specified (`-a` option):

```
hieradata -a ldap_servers fqdn=ldap.example42.com
["127.0.0.1", "10.42.10.31", "10.42.10.32"]
```

Note that the value defined for the `env=test` case is not returned, unless we specify it:

```
hiera -a ldap_servers fqdn=ldap.example42.com env=test
["127.0.0.1", "192.168.0.31", "10.42.10.31", "10.42.10.32"]
```

When working with hashes, interesting things can be done.

Let's see what the value of the user's hash is for the test environment:

```
hiera users env=test
{"qa"=>{"home"=>"/home/qa", "comment"=>"QA Tester"}}
```

As it normally does, Hiera returns the first value it meets while traversing the hierarchy, but in this case, we might prefer to have a hash containing all the values found. Similar to the `-a` option for arrays, we have the `-h` option for hashes at our disposal:

```
hiera -h users env=test
{"al"=>{"home"=>"/home/al", "comment"=>"Al"},
 "jenkins"=>{"password"=>"!", "comment"=>"Jenkins"},
 "qa"=>{"home"=>"/home/qa", "comment"=>"QA Tester"}}
```



Note that hashes are not ordered according to the matching order as arrays.

Let's make one further experiment, let's add a new user specific for our `ldap.example42.com` node and give different values to a parameter already defined in the common data source:

```
vi /etc/puppet/hieradata/nodes/ldap.example42.com.yaml

users:
  openldap:
    groups: 'apps'
  jenkins:
    ensure: absent
```

A hash lookup merges the found values as expected:

```
hiera -h users fqdn=ldap.example42.com env=test

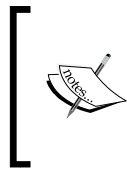
{"al"=>{"home"=>"/home/al", "comment"=>"Al"},
 "jenkins"=>{"ensure"=>"absent"},
 "qa"=>{"home"=>"/home/qa", "comment"=>"QA Tester"},
 "openldap"=>{"groups"=>"apps"}}
```

Let's take a look at the parameters of the `jenkins` user; being defined both at node level and in the common data source, the returned value is the one for the higher data source in the hierarchy.

Hiera's management of hashes can be quite powerful, and we can make optimal use of it. For example, we can use them inside Puppet manifests with the `create_resources` function, with the hash of the `users` data and a single line of code as follows:

```
create_resources(user, hiera_hash($users))
```

Based on highly customizable Hieradata, we can manage all the users of our nodes.



We can tune how Hieradata manages the merging of hashes with the `merge_behavior` global setting, which allows deeper merging at single key levels. Read the official documentation at http://docs.puppetlabs.com/hiera/1/lookup_types.html#hash-merge for more details.

Quite often, we need to understand where a given key is set in our hierarchy and what values will be computed for it. The `-d` (debug) option is rather useful for this. The previous line will return the following output:

```
hiera -d -h users fqdn=ldap.example42.com env=test
DEBUG: 2013-12-07 13:11:07 +0100: Hieradata YAML backend starting
DEBUG: <datetime>: Looking up users in YAML backend
DEBUG: <datetime>: Looking for data source nodes/ldap.example42.com
DEBUG: <datetime>: Found users in nodes/ldap.example42.com
DEBUG: <datetime>: Looking for data source env/test
DEBUG: <datetime>: Found users in env/test
DEBUG: <datetime>: Looking for data source common
DEBUG: <datetime>: Found users in common
{"al"=>{"home"=>"/home/al", "comment"=>"Al"},
```

```
"jenkins"=>{"ensure"=>"absent"},
"qa"=>{"home"=>"/home/qa", "comment"=>"QA Tester"},
"openldap"=>{"groups"=>"apps"}}
```

This output also tells us where Hiera is actually looking for data sources.

In a real Puppet environment, it is quite useful to use the `--yaml` option, which, used with a real facts file of a node, allows us to evaluate exactly how Hiera computes its keys for real servers.

On the Puppet Master, the facts of all the managed clients are collected in `$vardir/yaml/facts`, so this is the best place to see how Hiera evaluates keys for different clients:

```
hiera --yaml /var/lib/puppet/yaml/facts/<node>.yaml ldap_servers
```

Hiera can use other sources to retrieve the facts of a node and return its key value accordingly. We can interrogate the Puppet Master's inventory service with the following command:

```
hiera -i ldap.example42.com ldap_servers
```

We can query `mcollective` (from a machine where the `mco` client is installed):

```
hiera -m ldap.example42.com ldap_servers
```

Using Hiera in Puppet

The data stored in Hiera can be retrieved by the Puppet Master while compiling the catalog using the Hiera functions. In our manifests, we can have something like the following:

```
$dns_servers = hiera("dns_servers")
```

Note that the name of the Puppet variable need not be the same as the Hiera one, so the preceding command can also be something like this:

```
$my_dns_servers = hiera("dns_servers")
```

This assigns the top value to the `$my_dns_servers` variable (the first one found while crossing the hierarchy of data sources) retrieved by Hiera for the key `dns_servers`.

We can also merge arrays and hashes here, so, in order to retrieve an array of **all the values** in the hierarchy's data sources of a given key and not just the first one, we can use `hiera_array()`:

```
$my_dns_servers = hiera_array("dns_servers")
```

If we expect a hash value for a given key, we can use the `hiera()` function to retrieve the top value found, or `hiera_hash()` to merge all the found values in a single hash:

```
$openssh_settings = hiera_hash("openssh_settings")
```

All these Hiera functions may receive additional parameters, as follows:

- **Second argument:** If present and not blank, then this is the default value to use if no value is found for the given key.
- **Third argument:** This overrides the configured hierarchy adding a custom data source at the top of it. This might be useful in cases where we need to evaluate data using a logic not contemplated by the current hierarchy and for which it isn't worth the effort to add an extra layer in the global configuration.

The following code shows the usage of additional parameters:

```
$my_dns_servers = hiera("dns_servers", "8.8.8.8", "$country")
```

Dealing with hashes in Puppet code

With a hash, it is possible to express complex and structured data that has to be managed inside the Puppet code.

Remember that Hiera always returns the value of the first defined level keys, for example, we have a hash with nested hashes, as shown in the following code:

```
network::interfaces_hash:
  eth0:
    ipaddress: 10.0.0.193
    netmask: 255.255.255.0
    network: 10.0.0.0
    broadcast: 10.0.0.255
    gateway: 10.0.0.1
    post_up:
      - '/sbin/ifconfig eth3 up'
      - '/sbin/ifconfig eth4 up'
  eth2:
    enable_dhcp: true
  eth3:
```

```

    auto: false
    method: manual
  eth4:
    auto: false
    method: manual

```

We can create a variable inside our Puppet code that loads it:

```
$int_hash = hiera('network::interfaces_hash')
```

Then, refer to single values inside its data structure with the following code:

```
$ip_eth0 = $int_hash['eth0']['ipaddress']
```

If we need to access this value from a template, we can directly write it in our erb file:

```
ipaddress <%= @int_hash['eth0']['ipaddress'] %>
```



A complex hash like the preceding is typically used with a `create_resources` function as follows:

```
create_resources('network::interface', $interfaces_
hash)
```

Here the custom `network::interface` defined is expected to accept as argument a hash to configure one or more network interfaces.

Puppet 3 automatic parameter lookup

With Puppet 3 Hiera is shipped directly within the core code, but the integration goes far beyond: an automatic Hiera lookup is done for each class' parameter using the `$class::$argument` key; this functionality is called **data bindings** or **automatic parameter lookup**.

An example is the following class definition:

```

class openssh (
  $template = 'openssh/sshd.config.erb',
) { . . . }

```


The value of `$template` will be evaluated according to the following logic:

- If the user directly and explicitly passes the `template` argument, then its value is used:

```
class { 'openssh':  
  template => 'site/openssh/sshd_config.erb',  
}
```
- If no value is explicitly set, Puppet 3 automatically looks for the Hiera key `openssh::template`.
- Finally, if no value is found on Hiera, then the default `'openssh/sshd.config.erb'` is used.

To emulate a similar behavior on Puppet versions earlier than version 3, we would write something like the following:

```
class openssh (  
  $template = hiera("openssh::template", 'openssh/sshd.config.erb'),  
) { . . . }
```

Evolving usage patterns for class parameters

This strong integration has definitively boosted the adoption of Hiera and is changing the way Puppet code is organized and classes are declared.

Before Puppet 2.6, we could declare classes by just including them, optionally more than once in our catalog, using the following code:

```
include openssh
```

And, we could manage the behavior of the class just by setting variables and having them dynamically evaluated inside the class with something like the following code:

```
class openssh {  
  file { 'sshd_config':  
    content => template($openssh_template),  
  }  
}
```

This approach suffered the risks of having inconsistent values due to dynamic scoping of variables and parse ordering. Also, when using variables inside the module's code, it wasn't easy to understand which variables could affect the class's behavior, and there was not a public API, which is easily accessible.

The introduction of parameterized classes in Puppet 2.6 allowed classes to expose their arguments in a clear way using the following code:

```
class openssh (
  $template = 'openssh/sshd.config.erb',
) { . . . }
```

In order to pass them explicitly and consistently in a class declaration, use the following code:

```
class { 'openssh':
  template => 'site/openssh/sshd_config.erb',
}
```

But the fact that we can declare a specific class, using parameters, only **once** in a node's catalog presented new challenges on how our custom code had to be organized. We could not include classes wherever needed but we had to reorganize our manifests in order to avoid duplicate declarations.

From Puppet 3 onwards, it is possible to have the best of both worlds, we can use the original way to include classes:

```
include openssh
```

But we can also be sure that the `template` parameter is always and consistently evaluated according to the value of the Hiera key `openssh::template`.

Additional Hiera backends

The possibility of creating and adding different backends where we can store data is one of the strong points of Hiera, as it allows feeding Puppet with data from any possible source.

This allows integrations with existing tools and gives more options to provide data in a safe and controlled way, for example, a custom web frontend or a CMDB.

Let's review some of the most interesting backends that exist.

Hiera-file

Hiera-file (<https://github.com/adrienthebo/hiera-file>) has been conceived by Adrien Thebo to manage a kind of data that previously couldn't be stored in a sane way in Hiera – that is, plain files.

To install it, just clone the previous Git repository in our `modulepath` or use its `gem` as follows:

```
gem install hiera-file
```

We configure it by specifying a `datadir` path where our data files are placed:

```
---
:backends:
  - file
:hierarchy:
  - "fqdn/{fqdn}"
  - "role/{role}"
  - "common"
:file:
  :datadir: /etc/puppetlabs/code/data
```

Here, the key used for Hieradata lookups is actually the name of a file present in `.d` subdirectories inside our `datadir` according to our hierarchy.

For example, consider the following Puppet code:

```
file { ['/etc/ssh/sshd_config']:
  ensure => present,
  content => hieradata('sshd_config'),
}
```

Given the previous hierarchy (first file found is returned), the code will create a `/etc/ssh/sshd_config` file with the content taken from files searched in these places:

```
/etc/puppetlabs/code/data/fqdn/<fqdn>.d/sshd_config
/etc/puppetlabs/code/data/role/<role>.d/sshd_config
/etc/puppetlabs/code/data/common.d/sshd_config
```

In the preceding examples, `<fqdn>` and `<role>` have to be substituted with the actual FQDN and role of our nodes.

If we want to provide an ERB template using `hieradata-file` we can use this syntax:

```
file { ['/etc/ssh/sshd_config']:
  ensure => present,
  content => inline_template(hieradata('sshd_config.erb')),
}
```

This will look for an erb template and parse it from:

```
/etc/puppetlabs/code/data/fqdn/<fqdn>.d/ssh_config.erb
/etc/puppetlabs/code/data/role/<role>.d/ssh_config.erb
/etc/puppetlabs/code/data/common.d/ssh_config.erb
```

Hiera-file is quite simple to use and very powerful because it allows us to move to Hiera what is generally placed in (site) modules: plain files with which we manage and configure our applications.

Encryption plugins

Being Puppet code and data generally versioned with a SCM and distributed accordingly, it has always been an issue to decide how and where to store reserved data such as passwords, private keys, and credentials. They were generally values assigned to variables either in clear text or as MD5/SHA hashes, but the possibility to expose them has always been a concern for Puppeteers, and various more or less imaginative solutions have been tried (sometimes, the solution has been to just ignore the problem and have no solution).

Hiera-gpg

One of these solutions is the `hiera-gpg` (<https://github.com/crayfishx/hiera-gpg>) plugin, kept in this edition for historical reasons, as it can still be used in old Puppet versions even if it's now deprecated.

Install `hiera-gpg` via its gem (we also need to have the `gpg` executable in our PATH, `gcc` and the Ruby development libraries package (`ruby-devel`)):

```
gem install hiera-gpg
```

A sample `hiera.yaml` code is as follows:

```
---
:backends:
  - gpg
:hierarchy:
  - %env%
  - common
:gpg:
  :datadir: /etc/puppetlabs/code/gpgdata
#  :key_dir: /etc/puppet/gpg
```

The `key_dir` is where our `gpg` keys are looked for, if we don't specify it, they are looked by default in `~/ .gnupg`, so, on our Puppet Master, this would be the `.gnupg` directory in the home of the puppet user.

```
gpg --gen-key
```

We will be asked for the kind of key, its size and duration (default settings are acceptable), a name, an e-mail, and a passphrase (even if `gpg` will complain, do not specify a passphrase as `hieradata-bbox="171 255 799 288" data-label="Text">

Once the key is created, we can show it using the following command (eventually move the content of our ~/.gnupg to the configured key_dir):`

```
gpg --list-key
/root/.gnupg/pubring.gpg
-----
pub   2048R/C96EECCF 2013-12-08
uid                               Puppet Master (Puppet) <al@lab42.it>
sub   2048R/0AFB6B1F 2013-12-08
```

Now we can encrypt files, move into our `gpg` datadir, and create normal YAML files containing our secrets, for example:

```
---
mysql::root_password: 'V3rys3cr3T!'
```

Note that this is a temporary file that we will probably want to delete, because we'll use its encrypted version, which has to be created with the following command:

```
gpg --encrypt -o common.gpg -r C96EECCF common.yaml
```

The `-r` argument expects our key ID (as seen via `gpg --list-key`), and `-o` expects the output file, which must have the same name/path of our data source with a `.gpg` suffix.

Then we can finally use `hieradata-bbox="171 652 433 666" data-label="Text">

```
hieradata-bbox="201 677 816 810" data-label="Text">

```
DEBUG: <datetime>: Hieradata-bbox="171 839 516 854" data-label="Page-Footer">

[56]


```


````

```

DEBUG: <datetime>: [gpg_backend]: result is a String ctx
#<GPGME::Ctx:0x7fb6aaa2f810> txt ---

mysql::root_password: 'V3rys3cr3T!'

DEBUG: <datetime>: [gpg_backend]: GPG decrypt returned valid data

DEBUG: <datetime>: [gpg_backend]: Data contains valid YAML
DEBUG: <datetime>: [gpg_backend]: Key mysql::root_password found in
YAML document, Passing answer to hiera
DEBUG: <datetime>: [gpg_backend]: Assigning answer variable

```

Now we can delete the cleartext `common.yaml` file and safely commit in our repository the encrypted GPG file and use our public key for further edits.

When we need to edit our file, we can decrypt it with the following command:

```
gpg -o common.yaml -d common.gpg
```



Note that we'll need the `gpg` private key to decrypt a file; this is needed on the Puppet Master and we need it on any system where we intend to edit these files.

Hiera-gpg is a neat solution to manage sensitive data but it has some drawbacks, the most relevant one is that we have to work with full files and we don't have a clear control on who makes changes to, unless we distribute the `gpg` private key to each member of our team.

Hiera-eyaml

Hiera-eyaml is currently the most recommended plugin to manage secure data with `hiera`. It was created as an evolution of `hiera-gpg`, and has some improvements over the older plugin:

- It only encrypts the values, and does it individually, so files can be easily reviewed and compared
- It includes a tool to edit and view the files, so they can be used almost as easily as nonencrypted data files
- It also manages nonencrypted data, so it can be used as the only solution for YAML files
- It uses basic asymmetric encryption that reduces dependencies, but it also includes a pluggable framework that allows the addition of other encryption backends.

Let's see how `hier`-`eyaml` works, as it is more used and maintained than `hier`-`gpg`.

We install `gem`:

```
gem install hier-eyaml
```

We edit the `hier`.`yaml` to configure it:

```
---
:backends:
  - eyaml
:hierarchy:
  - "nodes/{fqdn}"
  - "env/{env}"
  - common
:eyaml:
  :datadir: /etc/puppet/code/hieradata
  :pkcs7_private_key: /etc/puppet/keys/private_key.pkcs7.pem
  :pkcs7_public_key: /etc/puppet/keys/public_key.pkcs7.pem
```

Now, at our disposal is the powerful `eyaml` command, which makes the whole experience pretty easy and straightforward. We can use it to create our keys, encrypt and decrypt files or single strings, and directly edit on-the-fly files with encrypted values:

1. First, let's create our keys using the following command:
`eyaml createkeys:`
2. They are placed in the `./keys` directory; make sure that the user under which the Puppet Master runs (usually `puppet`) has read access to the private key.
3. Now we can generate the encrypted value of any `hier` key with:
`eyaml encrypt -l 'mysql::root_password' -s 'V3ryS3cr3T!'`
4. Write a space before the command so it's not stored in bash history. This will print, on `stdout`, both the plain encrypted string and a block of configuration that we can directly copy in our `.eyaml` files:

```
vi /etc/puppet/hieradata/common.eyaml
---
mysql::root_password: >
    ENC[PKCS7,MIIBeQYJKoZIhvcNAQcDoIIBaJCCAAYCAQAxggEhMII  [...]
    +oefgBBdAJ60kXMMh/RHpaXQYX3T]
```

Note that the value is in this format: `ENC [PKCS7, Encrypted_Value]`.

Luckily, in a similar fashion, we have to generate the keys only once, since great things happen when we have to change our encrypted values in our `eyaml` files. We can directly edit them with the following command:

```
eyaml edit /etc/puppet/hieradata/common.eyaml
```

Our editor will open the file and we will see the decrypted values, as it will decrypt them on the fly, so that we can edit our secrets in clear text and save the file again (of course, we can do this only on a machine where we have access to the private key). The decrypted values will appear in the editor between brackets and prefixed by `DEC` and the encryption backend used, `PKCS7` by default:

```
mysql::root_password: DEC(1)::PKCS7[V3ryS3cr3T!!]
```

This makes the management and maintenance of our secrets particularly easy. To view the decrypted content of a `eyaml` file, we can use:

```
eyaml decrypt -f /etc/puppet/hieradata/common.eyaml
```

Since `hiera-eyaml` manages both clear text and encrypted values, we can use it as our only backend if we want to work only on YAML files.

Hiera-http, hiera-mysql

Hiera-http (<https://github.com/crayfishx/hiera-http>) and hiera-mysql (<https://github.com/crayfishx/hiera-mysql>) are other powerful Hiera backends written by Craig Dunn; they perfectly interpret Hiera's modular and extendable design and allow us to retrieve our data either via a REST interface or via MySQL queries on a database.

A quick view of how they could be configured can give an idea of how they can fit different cases. To configure `hiera-http` in our `hiera.yaml`, place something like this:

```
:backends:
  - http
:http:
  :host: 127.0.0.1
  :port: 5984
  :output: json
  :failure: graceful
  :paths:
    - /configuration/{fqdn}
    - /configuration/{env}
    - /configuration/common
```


To configure `hiera-mysql`, run the following code:

```
---
:backends:
  - mysql
:mysql:
  :host: localhost
  :user: root
  :pass: examplepassword
  :database: config
  :query: SELECT val FROM configdata WHERE var='%{key}' AND
environment='%{env}'
```

We will not examine them deeper and leave the implementation and usage details to the official documentation. Just consider how easy and intuitive the syntax to configure them and what powerful possibilities they open. They let users manage Puppet data from, for example, a web interface, without touching Puppet code or even logging in to a server and working with a SCM such as Git.

Using Hiera as an ENC

Hiera provides an interesting function called `hiera_include`, which allows you to include all the classes defined for a given key.

This, in practice, exploits the Hiera flexibility to provide classes to nodes as does an External Node Classifier.

It's enough to place in our `/etc/puppet/manifests/site.pp` a line like this:

```
hiera_include('classes')
```

Then, define in our data sources a `classes` key with an array of the classes to include.

In a YAML-based backend, it would look like the following:

```
---
classes:
  - apache
  - mysql
  - php
```

This is exactly the same as having something like the following in our `site.pp`:

```
include apache
include mysql
include php
```

The `classes` key (it can have any name, but `classes` is a standard de facto) contains an array, which is merged along the hierarchy. So, in `common.yaml`, we can define the classes that we want to include on all our nodes, and include specific classes for specific servers, adding them at the different layers of our hierarchy.

Along with the `hiera-file` backend, this function can help us have a fully Hierarchical-driven configuration on our Puppet architecture as we'll see in *Chapter 4, Designing Puppet Architectures*. It is one of the many options we have to glue together that define and build our infrastructure.

Summary

Hiera is a powerful and integrated solution to manage Puppet data outside our code. It requires some extra knowledge and abstraction, but it brings with it a lot of flexibility and extendibility, thanks to its backend plugins.

In this chapter, we have seen how it works and how to use it, especially when coupled with data bindings. We have also reviewed the most used plugins and grasped the power that comes with them.

Now is the time to explore another relatively new and great component of a Puppet infrastructure that is going to change how we use and write Puppet code — PuppetDB.

3

Introducing PuppetDB

A model based on agents that receive and apply a catalog received from the Puppet Master has an intrinsic limitation: the client has no visibility and direct awareness about the state of resources of the other nodes.

It is not possible, for example, to execute during the catalog application functions that do different things according to different external conditions. There are many cases where information about other nodes and services could be useful to manage local configurations, for example, we might:

- Need to start a service only when we are sure that the database, the queues, or any external resource it relies upon are already available in some external nodes
- Configure a load balancer that dynamically adds new servers, if they exist
- Have to manage the setup of a cluster which involves specific sequences of commands to be executed in a certain order on different nodes

The declarative nature of Puppet's DSL might look inappropriate to manage setups or operations where activities have to be done in a procedural way, which might be based on the availability of external resources.

Part of the problem can be solved using facts: being executed on the client, they provide direct information about its environment.

We will see in the next chapters how to write custom ones, but the basic concept is that they can contain the output of any command we may want to execute: checks of the state of applications, availability of remote services, system conditions, and so on.

We can use these facts in our manifests to define the resources to apply on the client and manage some of the above cases. Still, we cannot have, on a node, direct information about resources on other nodes, besides what can be checked remotely.

The challenge, or at least a part of it, has been tackled some years ago with the introduction of **exported resources**, as we have seen in *Chapter 1, Puppet Essentials*, they are special resources declared on one node but applied on another one.

Exported resources need the activation of the `storeconfigs` option, which used Rails' Active Record libraries for data persistence.

Active Record-based stored configs have served Puppet users for years, but they suffered from performance issues which could be almost unbearable on large installations with many exported resources.

In 2011, Deepak Giridharagopal, a Puppet Labs lead engineer, tackled the whole problem from a totally detached point of view and developed PuppetDB, a marvelous piece of software that copes not only with stored configs but also with all Puppet-generated data.

In this chapter, we will see:

- How to install and configure PuppetDB
- An overview of the available dashboards
- A detailed analysis of PuppetDB API
- How to use the **puppetdbquery** module
- How PuppetDB can influence our future Puppet code

Installation and configuration

PuppetDB is an Open Source Closure application complementary to Puppet. It does exactly what the name suggests: it stores Puppet data:

- All the **facts** of the managed nodes
- A copy of the **catalog** compiled by the Master and sent to each node
- The **reports** of the subsequent Puppet runs, with all the **events** that have occurred

What is stored can be queried, and for this PuppetDB exposes a REST-like API that allows access to all its data.

Out of the box, it can act as an alternative to two functions previously done using the Active Records libraries:

- The backend for stored configs, where we can store our exported resources
- A replacement for the **inventory service** (an API we can use to query the facts of all the managed nodes)

While read operations are based on a REST-like API, data is written by **commands** sent by the Puppet Master and queued asynchronously by PuppetDB to a pool of internal workers that deliver data to the persistence layer, based either on the embedded HSQLDB (usable mostly for testing or small environments) or on PostgreSQL.

On medium and large sites PuppetDB should be installed on dedicated machines (eventually with PostgreSQL on separated nodes); on a small scale it can be placed on the same server where the Puppet Master resides.

A complete setup involves:

- On the PuppetDB server: the configuration of the init scripts, the main configuration files, and logging
- On our Puppet server configuration directory: the connection settings to PuppetDB in `puppetdb.conf` and the `routes.yaml` file

Generally, communication is always between the Puppet Master and PuppetDB, based on certificates signed by the CA on the Master, but we can have a masterless setup where each node communicates directly with PuppetDB.



Masterless PuppetDB setups won't be discussed in this book; for details, check https://docs.puppetlabs.com/puppetdb/latest/connect_puppet_apply.html

Installing PuppetDB

There are multiple ways to install PuppetDB. It can be installed from source, from packages, or using the Puppet Labs `puppetdb` module. In this book we are going to use the latter approach, so we also practice the use of community plugins. This module will be in charge of deploying a PostgreSQL server and PuppetDB.

First of all, we have to install the `puppet` module and its dependencies from the Puppet forge; they can be directly downloaded from their source or using Puppet:

```
puppet module install puppetlabs-puppetdb
```

Once installed in our Puppet server, it can be used to define the catalog of our PuppetDB infrastructure, it can be defined in three different ways:

- Installing it in the same server as the Puppet server; in this case it's enough to add the following line to our Puppet master catalog:

```
include puppetdb
```

- Or with masterless mode:

```
puppet apply -e "include puppetdb"
```

This is fine for testing or small deployments, but for larger infrastructures we'll probably need other kinds of deployment to improve performance and availability.

- Another option is to install PuppetDB in a different node. In this case the node with PuppetDB must include the class to install the server, and the class to configure the database backend:

```
include puppetdb::server
include puppetdb::database::postgresql
```

- We also need to configure the Puppet server to use this instance of PuppetDB; we can also use the PuppetDB module for that:

```
class { 'puppetdb::master::config':
  puppetdb_server => $puppetdb_host,
}
```

We'll see more details about this option in this chapter.

- If the previous options are not enough for the scale of our deployment, we can also have the database in a different server, then this server has to include the `puppetdb::database::postgresql` class, parameterized with its external hostname or address in the `listen_addresses` argument, to be able to receive external connections. The `puppetdb::server` class in the node with the PuppetDB server will need to be parameterized with the address of the database node, using the `database_host` parameter for that.

We can also specify other parameters as the a version to install, what may be needed depending on the version of Puppet our servers are running:

```
class { 'puppetdb':
  puppetdb_version => '2.3.7-1puppetlabs1',
}
```

In any of these cases, once Puppet is executed, we'll have PuppetDB running in our server, by default on port 8080. We can check it by querying the version through its API:

```
$ curl http://localhost:8080/pdb/meta/v1/version
{
  "version" : "3.1.0"
}
```



The list of available versions and the APIs they implement is available at <http://docs.puppetlabs.com/puppetdb/>.
PuppetDB puppet module is available at <https://forge.puppetlabs.com/puppetlabs/puppetdb>

If something goes wrong, we can check the logs in `/var/log/puppetlabs/puppetdb/`.



If we use the Puppet Labs puppetdb module to set up our PuppetDB deployment, we can take a look at the multiple parameters and sub-classes the module has. More details about these options can be found at:

http://docs.puppetlabs.com/puppetdb/latest/install_via_module.html
<https://github.com/puppetlabs/puppetlabs-puppetdb>

PuppetDB configurations

The configuration of PuppetDB involves operations on different files, such as:

- The configuration file sourced by the init script, which affects how the service is started
- The main configuration settings, placed in one or more files
- The logging configuration

Init script configuration

In the configuration file for the init script (`/etc/sysconfig/puppetdb` on RedHat or `/etc/default/puppetdb` on Debian), we can manage Java settings such as `JAVA_ARGS` or `JAVA_BIN`, or PuppetDB settings such as `USER` (the user with which the PuppetDB process will run), `INSTALL_DIR` (the installation directory), `CONFIG` (the configuration file path or the path of the directory containing `.ini` files).



To configure the maximum Java heap size we can set `JAVA_ARGS="-Xmx512m"` (recommended settings are 128m + 1m for each managed node if we use PostgreSQL, or 1g if we use the embedded HSQLDB. Raise this value if we see `OutOfMemoryError` exceptions in logs).

To expose the JMX interface we can set `JAVA_ARGS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=1099"`.

This will open a JMX socket on port 1099. Note that all the JMX metrics are also exposed via the REST interface using the `/metric` namespace.

Configuration settings

In Puppet Labs packages, configurations are placed in various `.ini` files in the `/etc/puppetlabs/puppetdb/conf.d/` directory.

Settings are managed in different [sections]. Let's see the most important ones.

Application-wide settings are placed in the following section:

[global]

Here are defined the paths where PuppetDB stores its files (`vardir`), configures `log4j` (`logging-config`) and some limits on the maximum number of results that a resource or event query may return. If those limits are exceeded the query returns an error; these limits can be used to prevent overloading the server with accidentally big queries:

```
vardir = /var/lib/puppetdb # Must be writable by Puppetdb user
logging-config = /etc/puppetdb/log4j.properties
resource-query-limit = 20000
event-query-limit = 20000
```

All settings related to the commands used to store data on PuppetDB are placed in the following section:

[command-processing]

Of particular interest is the `threads` setting, which defines how many concurrent command processing threads to use (default value is `CPUs/2`). We can raise this value if our command queue (visible from the performance dashboard we will analyze later) is constantly larger than zero.

In such cases, we should also evaluate if the bottleneck may be on the database performance.

Other settings are related to the maximum disk space (in MB) that can be dedicated to persistent (`store-usage`) and temporary (`temp-usage`) ActiveMQ message storage and for how long messages not delivered have to be kept in a **Dead Letter Office** before being archived and compressed (`dlo-compression-threshold`). Valid units here, as in some other settings, are days, hours, minutes, seconds, and milliseconds:

```
threads = 4
store-usage = 102400
temp-usage = 51200
dlo-compression-threshold = 1d # Same of 24h, 1440m, 86400s
```

All the settings related to database connection are in the following section:

```
[database]
```

Here we define what database to use, how to connect to it, and some important parameters about data retention.

If we use the (default) HSQLDB backend, our settings will be as follows:

```
classname = org.hsqldb.jdbcDriver
subprotocol = hsqldb
subname = file:/var/lib/puppetdb/db/db;hsqldb.tx=mvcc;sql.syntax_
pgs=true
```

For a (recommended) PostgreSQL backend, we need something like the following:

```
classname = org.postgresql.Driver
subprotocol = postgresql
subname = //<HOST>:<PORT>/<DATABASE>
username = <USERNAME>
password = <PASSWORD>
```

On our PostgreSQL server, we need to create a database and a user for PuppetDB:

```
sudo -u postgres sh
createuser -DRSP puppetdb
createdb -E UTF8 -O puppetdb puppetdb
```

Also, we have to edit the `pg_hba.conf` server to allow access from our PuppetDB host (here, it is `10.42.42.30`, but it could be `127.0.0.1` if PostgreSQL and PuppetDB are on the same host):

```
# TYPE DATABASE USER CIDR-ADDRESS METHOD
host all all 10.42.42.30/32 md5
```

Given the above examples and a PostgreSQL server with IP `10.42.42.35`, the connection settings would be as follows:

```
subname = //10.42.42.35:5432/puppetdb
username = puppetdb
password = <the password entered with the createuser command>
```

If PuppetDB and PostgreSQL server are on separate hosts, we may prefer to encrypt the traffic between them. To do so we have to enable SSL/TLS on both sides.



For a complete overview of the steps required, refer to the official documentation: http://docs.puppetlabs.com/puppetdb/latest/postgres_ssl.html

Other interesting settings manage how often in minutes the database is compacted to free up space and remove unused rows (`gc-interval`). To enable automatic deactivation of nodes if they are not reporting, the `node-tt` variable can be used, it can have a time value expressed in `d`, `h`, `m`, `s`, or `ms`. To completely remove deactivated nodes if they still don't report any activity use the `node-purge-ttl` variable and the retention for reports (when stored) is controlled by `report-ttl`; the default is `14d`:

```
gc-interval = 60
node-ttl = 15d # Nodes not reporting for 15 days are deactivated
node-purge-ttl = 10d # Nodes purged 10 days after deactivation
report-ttl = 14d # Event reports are kept for 14 days
```



The `node-ttl` and `node-purge-ttl` settings are particularly useful in dynamic and elastic environments where nodes are frequently added and decommissioned. Setting them allows us to automatically remove old nodes from our PuppetDB and, if we use exported resources for monitoring or load balancing, definitely helps in keep PuppetDB data clean and relevant. Obviously `node-ttl` must be higher than our nodes' Puppet run interval.

Be aware, though, that if we have the (questionable) habit of disabling regular Puppet execution for manual maintenance, tests, or whatever reason, we may risk deactivating nodes that are still working.

Finally note that nodes' automatic deactivation or purging is done when there is database compaction, so the `gc-interval` parameter must always be set with smaller intervals.

Another useful parameter is `log-slow-statements` that defines the number of seconds after any SQL query is considered *slow*. Slow queries are logged but still executed:

```
log-slow-statements = 10
```

Finally, some settings can be used to fine-tune the database connection pool; we probably won't need to change the default values (in minutes):

```
conn-max-age = 60 # Maximum idle time
conn-keep-alive = 45 # Client-side keep-alive interval
conn-lifetime = 60 # The maximum lifetime of a connection
```

We can manage the HTTP settings (used both for the web performance dashboard, the REST interface, and the commands) in the following section:

```
[jetty]
```

To manage HTTP unencrypted traffic we just have to define the listening IP (`host`, default `localhost`) and port:

```
host = 0.0.0.0 # Listen on any interface (Read Note below)
port = 8080    # If not set, unencrypted HTTP access is disabled
```



Generally, the communication between Puppet Master and PuppetDB is via HTTPS (using certificates signed by the Puppet Master's CA). However, if we enable HTTP to view the web dashboard (which just shows usage metrics, which are not particularly sensible), be aware that the HTTP port can be used also to query and issue commands to PuppetDB (so it definitely should not be accessed by unauthorized users). Therefore, if we open HTTP access to hosts other than localhost, we should either proxy or firewall the HTTP port to allow access to authorized clients/users only.

This is not an uncommon case, since the HTTPS connection requires a client host SSL authentication, so is not usable (in a comfortable way) to access the web dashboard from a browser.

For HTTPS access some more settings are available to manage the listening address (`ssl-host`) and port (`ssl-port`), the path to the PuppetDB server certificate PEM file (`ssl-cert`), its private key PEM file (`ssl-key`), and the path of the CA certificate PEM file (`ssl-ca-cert`) used for client authentication. In the following example, the paths used are the ones of Puppet's certificates that leverage the Puppet Master's CA:

```
ssl-host = 0.0.0.0
ssl-port = 8081
ssl-key = /var/lib/puppet/ssl/private_keys/puppetdb.site.com.pem
ssl-cert = /var/lib/puppet/ssl/public_keys/puppetdb.site.com.pem
ssl-ca-cert = /var/lib/puppet/ssl/certs/ca.pem
```

The above settings have been introduced in PuppetDB 1.4, and if present, are preferred to the earlier (and now deprecated) parameters that managed SSL via Java keystore files.

We report here a sample configuration that uses them as reference; we may find them on older installations:

```
keystore = /etc/puppetdb/ssl/keystore.jks
truststore = /etc/puppetdb/ssl/truststore.jks
key-password = s3cr3t # Passphrase to unlock the keystore file
trust-password = s3cr3t # Passphrase to unlock the truststore file
```



To set up SSL configurations, PuppetDB provides a very handy script that does the right thing according to the PuppetDB version and, eventually, the current configurations. Use it and follow the onscreen instructions:

```
/usr/sbin/puppetdb-ssl-setup
```

Other optional settings define the allowed cipher suites (`cipher-suites`), SSL protocols (`ssl-protocols`), and the path of a file that contains a list of certificate names (one per line) of the hosts allowed to communicate (via HTTPS) with PuppetDB (`certificate-whitelist`). If not set, any host can contact the PuppetDB, given that its client certificate is signed by the configured CA.

Finally, in our configuration file(s), we can enable real-time debugging in the `[repl]` section. This can be enabled to modify the behavior of PuppetDB at runtime and is used for debugging purposes, mostly by developers, so it is disabled by default.

For more information, check <http://docs.puppetlabs.com/puppetdb/latest/repl.html>

Logging configuration

Logging is done via Log4j and is configured in the `log4j.properties` file under the `logging-config` settings. By default, informational logs are placed in `/var/log/puppetdb/puppetdb.log`. Log settings can be changed at runtime and be applied without restarting the service.

Configurations on the Puppet Master

If we used the Puppet Labs puppet module to install PuppetDB we can also use it to configure our puppet master so that it sends the information about the executions to PuppetDB. This configuration is done by the `puppetdb::master::config` class. As we have seen in other cases, we can execute this class by including it in the Puppet catalog for our server:

```
include puppetdb::master::config
```

Or by running masterless Puppet:

```
puppet apply -e "include puppetdb::master::config"
```

This will install the `puppetdb-termini` package, as well as set up the required settings:

- In `/etc/puppetlabs/puppet/puppet.conf`, the PuppetDB backend has to be enabled for `storeconfigs` and, optionally, `reports`:


```
storeconfigs = true
storeconfigs_backend = puppetdb
report = true
reports = puppetdb
```

- In `/etc/puppetlabs/puppet/puppetdb.conf` the server name and port of PuppetDB are served, and if the Puppet Master should serve the catalog to clients when PuppetDB is unavailable
- `soft_write_failure = true`: in this case, a catalog is created without exported resources and facts, catalog, and reports are not stored. This option should not be enabled when exported resources are used. Default values are as follows:

```
server_urls = https://puppetdb.example.com:8081
soft_write_failure = false
```

- In `/etc/puppetlabs/puppet/routes.yaml` the facts terminus has to be configured to make PuppetDB the authoritative source for the inventory service. Create the file if it doesn't exist and run `puppet config print route_file` to verify its path:

```
---
master:
  facts:
    terminus: puppetdb
    cache: yaml
```

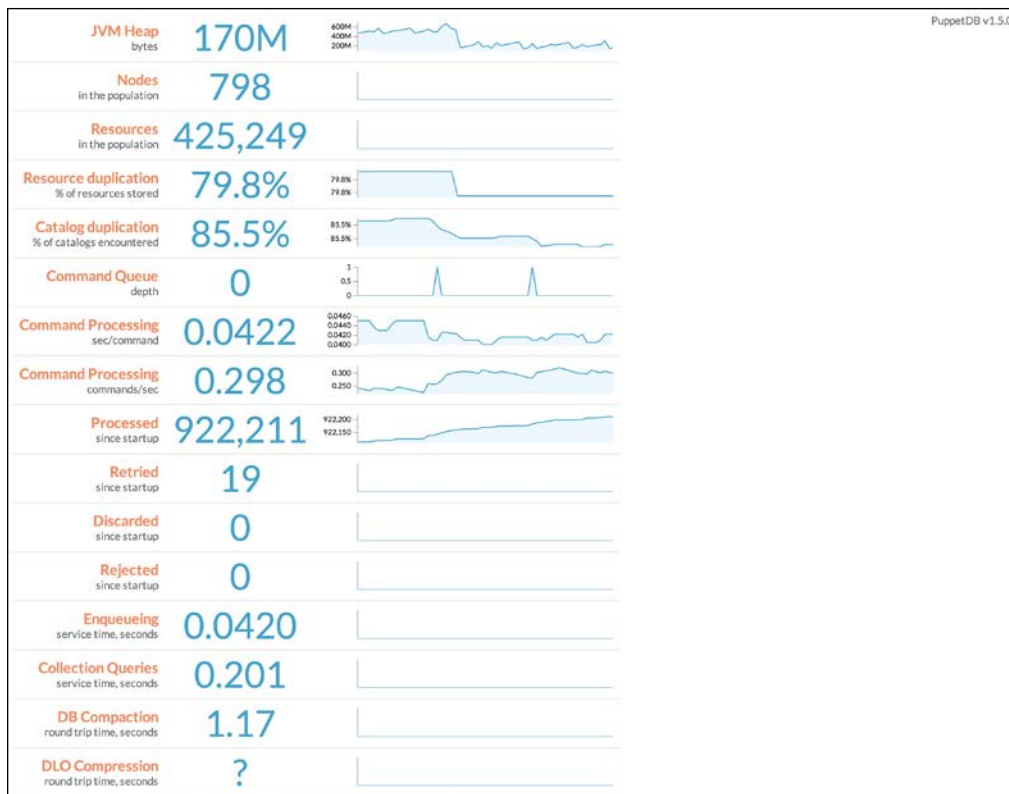
Dashboards

PuppetDB ecosystem provides web dashboards that definitely help user interaction:

- PuppetDB comes with an integrated performance dashboard
- Puppetboard is a web frontend that allows easy and direct access to PuppetDB data

PuppetDB performance dashboard

PuppetDB integrates a **performance dashboard** out of the box; we can use it to check how the software is working in real time. It can be accessed via HTTP at the URL `http://puppetdb.server:8080/pdb/dashboard/index.html` if you set `host = 0.0.0.0` on the PuppetDB configuration. Remember that you should limit HTTP access to unauthorized clients only, either by firewalling the host's port or setting `host = localhost` and having a local reverse proxy where you can manage access lists or authentication:



The PuppetDB performance dashboard

From the previous picture, the most interesting metrics are as follows:

- **JVM Heap memory usage:** It drops when the JVM runs a garbage collection.
- **Nodes:** The total number of nodes whose information is stored on PuppetDB.
- **Resources:** The total number of resources, present in all the catalogs stored.
- **Catalog duplication:** How much the stored catalogs have in common.
- **Command queue:** How many commands are currently in the queue; this value should not be constantly greater than a few units.
- **Command processing:** How many commands are delivered per second.
- **Processed:** How many commands have been processed since the service started.
- **Retried:** How many times commands submission has been retried since startup. A retry can be due to temporary reasons. A relatively low figure here is physiological; if we see it growing, we are having ongoing problems.

- **Discarded:** How many commands have been discarded since startup after all the retry attempts. Should not be more than zero.
- **Rejected:** How many commands were rejected and delivery failed since startup. Should not be more than zero.

Puppetboard—Querying PuppetDB from the Web

The amount of information stored on PuppetDB is huge and precious, and while it can be queried from the command line, a visual interface can help users explore Puppet's data.

Daniele Sluijters, a community member, started a project that has quickly become the visual interface of reference for PuppetDB: **Puppetboard** is a web frontend written in Python that allows easy browsing of nodes' facts, reports, events, and PuppetDB metrics.

It also allows us to directly query PuppetDB, so all the example queries from the command-line we'll see in this chapter can be issued directly from the web interface.

This is a relatively young and quite dynamic project that follows PuppetDB's APIs evolution; check its GitHub project for the latest installation and usage instructions: <https://github.com/nedap/puppetboard>.

PuppetDB API

PuppetDB uses a **Command/Query Responsibility Separation (CQRS)** pattern:

- Read activities are done for queries on the available REST, such as endpoints
- Write commands to update catalog, facts, and reports, and deactivate nodes

APIs are versioned (v1, v2, v3...). The most recent ones add functionalities and try to keep backwards compatibility.

Querying PuppetDB (read)

The URL for queries is structured like this:

```
http[s]://<server>:<port>/pdb/query/<version>/<endpoint>?query=<query>
```

Available **endpoints** for queries are: nodes, environments, factsets, facts, fact-names, fact-paths, fact-contents, catalogs, edges, resources, reports, events, event-counts, aggregate-event-counts, metrics, server-time, and version.

Query strings are URL-encoded JSON arrays in prefix notation, which makes them look a bit unusual. The general format is as follows:

```
[ "<operator>" , "<field>" , "<value>" ]
```

The comparison operators are: =, >=, >, <, <= and ~ (regexp matching). Some examples are as follows:

```
[ "=", "type", "Service"]
[">=", "timestamp", "2013-12-18T14:00:00"]
["~", "certname", "www\\d+\\.example\\.com"]
```

The expressions can be combined with `and`, `not`, and `or`. An example (here split over multiple lines for clarity) is as follows:

```
[ "and",
  ["=", "type", "File"],
  ["=", "title", "/etc/hosts" ]
]
```

It's possible to build complex **subqueries** using the `in` operator, the `extract` statement, and subqueries such as `select-resources` or `select-facts`. An example usable on the `/facts` endpoint to return the IPs of all the nodes that have an Apache service is as follows:

```
[ "and",  
  ["=", "name", "ipaddress"],  
  ["in", "certname",  
    ["extract", "certname",  
      ["select-resources",  
        ["and",  
          ["=", "type", "Service"],  
          ["=", "title", "apache"] ] ] ] ] ] ]
```

Since version 3 of API, it has been possible to paginate and sort the results of queries. Each endpoint may support one or more **query parameters**: `order-by`, `limit`, `include-total`, `offset`, and so on.

It's quite easy to query PuppetDB directly with `curl`; following is the simplest example, with `curl` executed on HTTP on the same PuppetDB host:

```
curl http://localhost:8080/pdb/query/v4/nodes/web01.example.com
```

Note the URL to a specific endpoint (`facts`), the API version (`v4`), and the specific client certname.

When we have to use queries, we must URL encode characters such as `[` and `]`, and for this we can use curl's `-data-urlencode` option. When we use it, we have to specify to use the `-X GET` option (otherwise a POST would be done):

```
curl -X GET 'http://localhost:8080/pdb/query/v4/events' -data-urlencode 'query=["=", "certname", "puppet.example.com"]'
```

The response, in JSON array format (note the starting and ending square brackets `[ ]`), contains one or more entries like this:

```
[ {  
  
  "new_value" : "{md5}be99db88f4c07058843ea356eb3469bf",  
  
  "report" : "2331579061f83db1a35e7579a83a671f011e07fa",  
  
  "run_start_time" : "2016-03-19T21:17:26.790Z",  
  
  "property" : "content",  
  
  "file" : "/etc/puppetlabs/code/environments/production/modules/  
puppetdb/manifests/master/routes.pp",  
  
  "old_value" : "{md5}d13e1f5c099082afbe8a5ed9d4695beb",  
  
  "containing_class" : "Puppetdb::Master::Routes",  
  
  "line" : 38,  
  
  "resource_type" : "File",  
  
  "status" : "success",  
  
  "configuration_version" : "1458422249",  
  
  "resource_title" : "/etc/puppetlabs/puppet/routes.yaml",  
  
  "environment" : "production",  
  
  "timestamp" : "2016-03-19T21:17:39.138Z",  
}
```

```

"run_end_time" : "2016-03-19T21:17:36.705Z",

"report_receive_time" : "2016-03-19T21:18:38.350Z",

"containment_path" : [ "Stage[main]", "Puppetdb::Master::Routes",
"File[/etc/puppetlabs/puppet/routes.yaml]" ],

"certname" : "puppet.example.com",

"message" : "content changed '{md5}d13e1f5c099082afbe8a5ed9d4695beb'
to '{md5}be99db88f4c07058843ea356eb3469bf'"

} ]

```



Have a look at some of the most interesting fields: timestamp, certname, resource-title, resource-type, property, file and line. Note that the name and kind of the fields may vary according to the endpoint used (for example, on other endpoints we have title and type instead of resource-title and resource-type).

It's recommended to experiment with test queries on various endpoints, such as the ones listed later in this chapter, to have a better idea of the kind and name of fields returned.

When we make requests over HTTPS we have to reference the certificates' files to use:

```

$ curl 'https://puppetdb:8081/pdb/query/v4/facts/web01.example.com' \
--cacert /var/lib/puppet/ssl/certs/ca.pem \
--cert /var/lib/puppet/ssl/certs/<node>.pem \
--key /var/lib/puppet/ssl/private_keys/<node>.pem

```

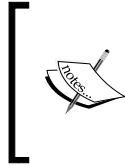
PuppetDB commands (write)

Explicit **commands** are used (via HTTP URL-encoded POST to the /commands URL) to populate and modify data.

The available commands on PuppetDB are:

- `replace catalog`: Replaces the stored catalog of a node. Currently PuppetDB stores only the last catalog compiled by the Puppet Master for each node.
- `replace facts`: Replaces the stored facts of a node. Also, in this case, only the ones received from the latest Puppet run are kept.

- `store report`: Saves the last report of a node's Puppet run (if reporting to PuppetDB is enabled). The configuration parameter `report-ttl` manages their retention (by default 14 days).
- `deactivate node`: Deactivates a decommissioned node so that its exported resources can't be collected anywhere. A node is reactivated if a new Puppet run is done on it.



This is the PuppetDB command done when we run on the Puppet Master: `puppet node deactivate <certname>`. Automatic deactivation of not reporting nodes can also be done via the `node-ttl` configuration option.

On `/var/log/puppetdb/puppetdb.log`, all the executed commands are shown.

When the Puppet Master receives a client's facts, it immediately submits them to PuppetDB:

```
2016-03-19 21:23:14,780 INFO [p.p.command] [51ab082d-a04f-4b11-a88e-ab38adc248d7] [replace facts] web01.example.com
```

Then the catalog is compiled, sent to the client, and stored on PuppetDB:

```
2016-03-19 21:23:26,050 INFO [p.p.command] [076e6a53-5d92-44a3-a550-05d9a99114fe] [replace catalog] web01.example.com
```

Finally, when the report of the Puppet run is received from the client, the Puppet Master submits it to PuppetDB:

```
2016-03-19 21:23:31,247 INFO [p.p.command] [ceff648b-f67d-44db-89c5-e0f9d1e936c4] [store report] puppet v4.3.1 - web01.example.com
```

Querying PuppetDB for fun and profit

PuppetDB stores and exposes a large amount of information. What can we do with it? Probably much more than what we might guess now. In this section, we explore in detail the REST endpoints available.

Diving into such details might be useful to better understand what can be queried and maybe trigger new ideas on what we can do with such information.

In these samples, we use `curl` with HTTP directly from the server where PuppetDB is installed.

/facts endpoint

Show all the facts of all our nodes (be careful, there may be a lot!):

```
curl 'http://localhost:8080/pdb/query/v4/facts'
```

Show the IP addresses of all our nodes (a similar search can be for any fact):

```
curl 'http://localhost:8080/pdb/query/v4/facts/ipaddress'
```

Show the node that has a specific IP address:

```
curl 'http://localhost:8080/pdb/query/v4/facts/ipaddress/10.42.42.27'
```

Show all the facts of a specific node:

```
curl -X GET http://localhost:8080/pdb/query/v4/facts \
--data-urlencode 'query=["=", "certname", "web01.example.com"]'
```

The response is always a JSON array with an entry per fact. Each entry is like the following:

```
{ "certname": <node name>, (IE: www01.example.com)
  "name": <fact name>, (IE: operatingsystem)
  "value": <fact value> (IE: ubuntu) }
```

/resources endpoint

Show all the resources of type Mount for all the nodes:

```
curl 'http://localhost:8080/pdb/query/v4/resources/Mount'
```

Note that the resource type must be capitalized, as we are referring to the type, and not to an specific instance.

Show all the resources of a given node:

```
curl -X GET http://localhost:8080/pdb/query/v4/resources --data-urlencode \
'query=["=", "certname", "web01.example.com"]'
```

Show all nodes that have Service['apache'] with ensure = running:

```
curl -X GET http://localhost:8080/pdb/query/v4/resources/Service \
--data-urlencode 'query=[ "and" , ["=", "title", "apache" ],
                        ["=", ["parameter", "ensure"], "running"] ]'
```

Same as before, using a different approach:

```
curl -X GET http://localhost:8080/pdb/query/v4/resources/Service/apache \
--data-urlencode 'query=["=", ["parameter", "ensure"], "running"]'
```

Show all the resources managed for a given node in a given manifest:

```
curl -X GET http://localhost:8080/pdb/query/v4/resources/ --data-
urlencode \
'query=["and" ["=", "file", "/etc/puppet/manifests/apache.pp"], \
["=", "certname", "web01.example.com"]]'
```

The response format of the resources endpoint shows how we can query everything about the resources managed by Puppet and defined in our manifests:

```
{
  "certname": " <node name>", (IE: www01.example.com)
  "resource": " <the resource's unique hash>", (IE: f3h34ds...)
  "type": " <resource type>", (IE: Service)
  "title": " <resource title>", (IE: apache)
  "exported": " <true|false>", (IE: false)
  "tags": [" <tag>", " <tag>"], (IE: "apache", "class" ...)
  "file": " <manifest path>", (IE: "/etc/puppet/manifests/site.pp")
  "line": " <manifest line>", (IE: "3")
  "parameters": {
    <parameter>: <value>, (IE: "enable" : true,)
    <parameter>: <value>,
    ...}}
}
```

/nodes endpoint

Show all the (not deactivated) nodes:

```
curl 'http://localhost:8080/pdb/query/v4/nodes'
```

Show all the facts of a specific node (this is a better alternative than the earlier example):

```
curl 'http://localhost:8080/pdb/query/v4/nodes/www01.example.com/facts'
```

Show all the resources of a specific node (this is a better alternative than the earlier example):

```
curl 'http://localhost:8080/pdb/query/v4/nodes/www01.example.com/
resources'
```

Show all the nodes with the operating system CentOS:

```
curl -X GET http://localhost:8080/pdb/query/v4/nodes --data-urlencode \
'query=["=", ["fact", "operatingsystem"], "CentOS"]'
```

The response format is as follows:

```
{ "certname": <string>,
  "deactivated": <timestamp or null>,
  "expired": <timestamp or null>,
  "catalog_timestamp": <timestamp or null>,
  "facts_timestamp": <timestamp or null>,
  "report_timestamp": <timestamp or null>,
  "catalog_environment": <string or null>,
  "facts_environment": <string or null>,
  "report_environment": <string or null>,
  "latest_report_status": <string>,
  "latest_report_hash": <string>
}
```

When using the facts and resources sub URLs, we get replies in the same format as the relative endpoint.

/catalogs endpoint

Get the whole catalog (the last saved one) of a node (all the resources and edges):

```
curl 'http://localhost:8080/pdb/query/v4/catalogs/www01.example.com'
```

/fact-names endpoint

Get the names (just the names, not the values) of all the stored facts:

```
curl 'http://localhost:8080/pdb/query/v4/fact-names'
```

/metrics endpoint

These are mostly useful to check PuppetDB performances and operational statistics. Some of them are visible from the performance dashboard.

Get the names of all the metrics available:

```
curl 'http://localhost:8080/metrics/v1/mbeans'
```

The result shows a remarkable list of items in JMX Mbean ObjectName style:

```
<Mbean-doman>:type=<Type>[,name:<Name>]
```

An example, in URL-encoded format as returned by PuppetDB, is as follows:

```
"com.jolbox.bonecp:type=BoneCP" : "/metrics/mbean/com.jolbox.bonecp%3Atype%3DBoneCP"
```


Available metrics are about nodes' population, database connection, delivery status of the processed commands, HTTP access hits, command processing, HTTP access, storage operations, JVM statistics, and the message queue system.

The following are few examples.

The total number of nodes in the population:

```
curl http://localhost:8080/metrics/v1/mbeans/com.puppetlabs.puppetdb.  
query.population%3Atype%3Ddefault%2Cname%3Dnum-nodes
```

The average number of resources per node:

```
curl http://localhost:8080/metrics/v1/mbeans/com.puppetlabs.puppetdb.  
query.population%3Atype%3Ddefault%2Cname%3Davg-resources-per-node
```

Statistics about the time used for the command `replace-catalog`:

```
curl http://localhost:8080/metrics/mbeans/com.puppetlabs.puppetdb.  
scf.storage%3Atype%3Ddefault%2Cname%3Dreplace-catalog-time
```

/reports endpoint

Show the summaries of all the saved reports of a given node:

```
curl -X GET http://localhost:8080/pdb/query/v4/reports --data-urlencode \  
'query=[ "=", "certname", "db.example.com" ] '
```

/events endpoint

Search all reports for failures:

```
curl -X GET 'http://localhost:8080/pdb/query/v4/events' --data-urlencode \  
'query=[ "=", "status", "failure" ] '
```

Search all reports for failures on Service type:

```
curl -X GET 'http://localhost:8080/pdb/query/v4/events' --data-urlencode \  
'query=[ "and", [ "=", "status", "failure" ], \  
[ "=", "resource-type", "Service" ] ] '
```

Search all reports for any change to the file with title *hosts*:

```
curl -X GET 'http://localhost:8080/pdb/query/v4/events' --data-urlencode \  
'query=[ "and", [ "=", "resource-type", "File" ], \  
[ "=", "resource-title", "hosts" ] ] '
```

Search all reports for changes in the content of the file with title *hosts*:

```
curl -X GET 'http://localhost:8080/pdb/query/v4/events' --data-urlencode \
  'query=[ "and", ["=", "resource-type", "File"], \
            ["=", "resource-title", "hosts" ], \
            ["=", "property", "content"] ]'
```

Show changes to the specified file only after a given timestamp:

```
curl -X GET 'http://localhost:8080/pdb/query/v4/events' --data-urlencode \
  'query=[ "and", ["=", "resource-type", "File"], \
            ["=", "resource-title", "hosts" ], \
            [ ">", "timestamp", "2015-12-18T14:00:00" ] ]'
```

Show all changes in a timestamp range:

```
curl -X GET 'http://localhost:8080/pdb/query/v4/events' --data-urlencode \
  'query=[ "and", [ ">", "timestamp", "2015-12-18T14:00:00" ] , \
            [ "<", "timestamp", "2015-12-18T15:00:00" ] ]'
```

Show all the changes related to resources provided by a specific manifest file:

```
curl -X GET 'http://localhost:8080/pdb/query/v4/events' --data-urlencode \
  'query=["=", "file", "/etc/puppet/modules/hosts/manifests/init.pp"]'
```

/event-counts endpoint

Show the count of resources of type *Service*, summarized per resource:

```
curl -X GET 'http://localhost:8080/pdb/query/v4/event-counts' \
  --data-urlencode 'query=["=", "resource-type", "Service" ]' \
  --data-urlencode 'summarize-by=resource'
```

Show the count of resources of type *Package*, summarized per node name:

```
curl -X GET 'http://localhost:8080/pdb/query/v4/event-counts' \
  --data-urlencode 'query=["=", "resource-type", "Package" ]' \
  --data-urlencode 'summarize-by=certname'
```

/aggregated-event-counts endpoint

Show the aggregated count of events for a node:

```
curl -G 'http://localhost:8080/pdb/query/v4/aggregate-event-counts' \
  --data-urlencode 'query=["=", "certname", "db.example.com"]' \
  --data-urlencode 'summarize-by=containing-class'
```

Show the aggregated count for all events on services on any node:

```
curl -G 'http://localhost:8080/pdb/query/v4/aggregate-event-counts' \
  --data-urlencode 'query=["=", "resource-type", "Service"]' \
  --data-urlencode 'summarize-by=certname'
```

/server-time endpoint

Show PuppetDB server's time, in ISO-8601 format (the format we'll deal with when querying timestamps):

```
curl http://localhost:8080/pdb/query/v4/server-time
```

/version endpoint

Show PuppetDB's version:

```
curl http://localhost:8080/pdb/query/v4/version
```

The puppetdbquery module

By now, we have realized how comprehensive the amount of information stored on PuppetDB is, as it provides a complete view of all our nodes facts, catalogs, and reports. This is useful for a review of what happens on our infrastructure and for the metrics we can extract via queries on all the resources managed by Puppet, but that's not enough.

One of Puppet's limitations, the fact that a node basically has knowledge only about itself via its catalog, and can "interact" with other nodes only via exported resources, would be wiped out if it were possible to make all PuppetDB data at our disposal when compiling a catalog for a node.

Well, this is possible, and can be easily done via Eric Dalén's `puppetdbquery` module.

Consider it the key that opens PuppetDB wonders to our Puppet code. It provides the following:

- Command line tools (as a Puppet face)
- Functions to query PuppetDB directly in our manifests
- A Hieradata backend.

This module enables PuppetDB integration to the next level.

We can install it from the Forge:

```
puppet module install dalen-puppetdbquery
```

Query format

The `puppetdbquery` module uses a custom format for queries which is different (and easier to use) from the native one. All the queries we can do with this module are in the following format:

```
Type[Name] {attribute1=foo and attribute2=bar}
```

By default, they are made on normal resources, using the `@@` prefix to query exported resources.

The comparison operators are: `=`, `!=`, `>`, `<` and `~` (regex matching).

The expressions can be combined with `and`, `not`, and `or`.

Querying from the command line

The module introduces, as a Puppet face, the `query` command that allows direct interaction with PuppetDB from the command line, for inline help:

```
puppet help query
```

To search for all the RedHat family nodes with version 6 we can type the following:

```
puppet query nodes '(osfamily=RedHat and lsbmajdistrelease=6)'
```

The same query done on facts shows all the facts for the resulting nodes:

```
puppet query facts '(osfamily=RedHat and lsbmajdistrelease=6)'
```

To show only the IP address of the queried nodes:

```
puppet query facts --facts ipaddress '(osfamily=RedHat and  
lsbmajdistrelease=6)'
```

Querying from Puppet manifests

The functions provided by the module can be used inside manifests to populate the catalog with data retrieved from PuppetDB.

`query_nodes` has two arguments: the query to use and (optional) the fact to return (by default it provides the `certname`). It returns an array, as follows:

```
$webservers = query_nodes('osfamily=Debian and Class[Apache]')
$webserver_ip = query_nodes('osfamily=Debian and Class[Apache]',
  ipaddress)
```

`query_facts` requires two arguments too: the query to use to discover nodes and the list of facts to return for them. It returns a nested hash in JSON format, as follows:

```
query_facts('Class[Apache]{port=443}', ['osfamily', 'ipaddress'])
```

These functions are dramatically useful to retrieve data from PuppetDB and provide resources on a node according to resources in catalogs compiled for other nodes.

The PuppetDB Hiera backend

Another powerful feature of the `puppetdbquery` module is the presence of a Hiera backend that allows us to use PuppetDB data for our Hiera keys.

It requires at least one other backend, so it's configured as follows:

```
---
:backends:
  - yaml
  - puppetdb

:hierarchy:
  - nodes/{fqdn}
  - common
```

The fun begins when we can use query in our keys. Instead of something like:

```
ntp::servers:
  - 'ntp1.example.com'
  - 'ntp2.example.com'
```

We can have a dynamic query like:

```
ntp::servers::_nodequery: 'Class[Ntp::Server]'
```

This returns an array with the certname of all the nodes in our infrastructure that have the `ntp::server` class. If we want their IP addresses, instead (the same applies for any other fact) use:

```
ntp::servers::_nodequery: ['Class[Ntp::Server]', 'ipaddress']
```

The above can also be written with this format (the result is the same):

```
ntp::servers::_nodequery:
  query: 'Class[Ntp::Server]'
  fact:  'ipaddress'
```

How Puppet code may change in the future

Now, hold on, stop to read and think again about what we've seen in this chapter, and particularly in the last section: variables that define our infrastructure can be dynamically populated according to the number of hosts that have specific classes or resources. If we add hosts with these services, they can be automatically used by the other hosts.

This is what we need to configure with Puppet dynamic and elastic environments where new services are made available to other nodes which are consequently configured.

For example, to manage a load balancer configuration, we can use, in the ERB template that is used for its configuration, a variable that returns all the IP addresses of the nodes that have Apache installed:

```
$web_servers_ip = query_nodes('Class[apache]', ipaddress)
```

This is a simple case that probably doesn't fit real scenarios where we probably have different Apache web servers doing different works in different servers, but it can give us an idea.

In other cases, we might need to know the value of a fact of a given node and use it on another node. In the following example, `cluster_id` might be a custom fact that returns an ID generated on the `db01` host; its value might be used on another host (a cluster member) to configure it accordingly:

```
$shared_cluster_id = query_facts('hostname=db01', cluster_id)
```

It's important to understand the kind of data we find and query on PuppetDB: what we find with the `puppetdbquery` module, for example, are resources contained in the last catalog generated by the Puppet Master and stored for each node. We are not sure if these resources have been applied successfully (we should query the events endpoint for that but currently that's not possible with this module) and we are not sure that the services expected are available.

Also consider how frequently Puppet runs on our nodes, as its *convergence time* may vary: if the interval is too large the infrastructure may adapt slowly to new changes; if it's too small we may have a greater risk of race conditions where a catalog that exposes a new service for a node has not yet been applied and, in the meantime, is already used to configure other nodes to use a service which might not already be configured.

These are probably hypothetical edge cases that we can tackle and manage in the same ways we would manage a temporarily faulty service, for example, excluding from a load-balancing pool non-responsive servers. Just be aware of them.

Summary

In this chapter, the word PuppetDB has been used zillions of times as an obsessive mantra. While we can use Puppet without it, as we have done for years, it's important to realize that PuppetDB is going to be present in every relevant Puppet infrastructure, and we can bet more and more applications and tools will emerge around it.

The fact that it's a robust piece of software engineered in a brilliant way makes us feel comfortable about the idea that Puppet Labs has decided to use it as central point of consolidation and gathering for all the data generated by Puppet.

We have seen how to configure PuppetDB and its integration with the Puppet Master, and how to interpret its performance dashboard. We have explored the principles of PuppetDB CQRS API, with REST-like endpoints for queries and commands for writing and, in some detail, the list of available endpoints, with various sample queries.

Finally, we have seen how most of the information gathered by PuppetDB can be queried from our manifests using the `puppetdbquery` module, and how this can dramatically influence how we manage interactions among different nodes.

Now that we grasp firmly the principles of Puppet, Hiera, and PuppetDB, we can explore how they can be glued together. In the next chapter, we will see how to deploy different architectures with them.

4

Designing Puppet Architectures

Puppet is an extensible automation framework, a tool, and a language. We can do great things with it and we can do them in many different ways. Besides the technicalities of learning the basics of its DSL, one of the biggest challenges for new and not-so-new users of Puppet is how to organize code and put things together in a manageable and appropriate way.

It's hard to find comprehensive documentation on how to use public code (modules), custom manifests and custom data, where to place our logic, how to maintain and scale it, and, generally, how to manage, safely and effectively, the resources that we want in our nodes and the data that defines them.

There's not really a single answer that fits all cases; there are best practices and recommendations, but ultimately it all depends on our own needs and infrastructure, which vary according to multiple factors. One of these principal factors is the characteristics of the infrastructure to manage itself, its size, and the number of application stacks to manage. Other factors are more related with Puppet code management, such as the skills of the people working with it, the number of teams involved, the integration with other tools, or the presence of policies for changes in production.

In this chapter, we will outline the elements needed to design a Puppet architecture, reviewing the following in particular:

- The **tasks** to deal with (manage nodes, data, code, files, and so on) and the available **components** to manage them
- The **Foreman** is probably the most used ENC around, along with Puppet enterprise console
- The **roles** and **profiles** pattern

- **Data separation** challenges and issues
- How the various components can be used together in different ways, with some **sample setups**

Components of a Puppet architecture

With Puppet, we manage our systems via the catalog that the Puppet Master compiles for each node, which is the total of the resources we have declared in our code, based on parameters and variables whose values reflect our logic and needs.

Most of the time, we also provide configuration files either as static files or viB templates, populated according to the variables we have set.

We can identify the following major **tasks** when we have to manage what we want to configure on our nodes:

- Definition of the classes to be included in each node
- Definition of the parameters to use for each node
- Definition of the configuration files provided to the nodes

These tasks can be provided by different, partly interchangeable, **components**:

- `site.pp`, the first file parsed by the Puppet Master and eventually all the files imported from there (import `nodes/*.pp` would import and parse all the code defined in the files with `.pp` suffix in the `/etc/puppet/manifests/nodes/` directory). Here we have code in Puppet language.
- An **External Node Classifier (ENC)** is an alternative source which can be used to define classes and parameters to apply to nodes. It's enabled with the following lines on the Puppet Master's `puppet.conf`:

```
[master]
node_terminus = exec
external_nodes = /etc/puppet/node.rb
```

What's referred by the `external_nodes` parameter can be any script that uses any backend; it's invoked with the client's `certname` as first argument (for example, `/etc/puppet/node.rb web01.example.com`) and should return a YAML-formatted output that defines the classes to include for that node, the parameters, and the Puppet environment to use.

Besides the well-known Puppet-specific ENC's, such as the **Foreman** and **Puppet Dashboard** (a former Puppet Labs project now maintained by community members), it's not uncommon to write custom ones that leverage on existing tools and infrastructure management solutions:

- **LDAP** can be used to store nodes information (classes, environment, and variables) as an alternative to the usage of an ENC. To enable LDAP integration, add the following to the Master's `puppet.conf`:

```
[master]
node_terminus = ldap
ldapserver = ldap.example.com
ldapbase = ou=Hosts,dc=example,dc=com
```

Then we have to add Puppet's schema to our LDAP server. For more information and details, check: http://docs.puppetlabs.com/guides/ldap_nodes.html

- **Hiera** is the hierarchical key-value data store we discussed in *Chapter 2, Managing Puppet Data with Hiera*. It's embedded in Puppet from version 3 and is available as an add-on on previous versions. Here we can set parameters, but also include classes and eventually provide contents for files.
- **Public** modules can be retrieved from the Puppet Forge, GitHub, or other sources; they typically manage applications and system settings. Being public, they might not fit all our custom needs, but they are supposed to be reusable, support different OS, and adapt to different usage cases. We are supposed to use them without any modification, as if they were public libraries, committing back to the upstream repository our fixes and enhancements. A common, but less recommended alternative is to fork a public module and adapt it to our needs. This might seem a quicker solution, but doesn't definitively help the open source ecosystem and would prevent us from having benefits from updates on the original repository.
- **Site module(s)** are custom modules with local resources and files where we can place all the logic we need, or the resources we can't manage, with public modules. They may be one or more, and may be called `site` or have the name of our company, customer, project, or anything in general. Site modules have particular sense as companions to public modules when used without local modifications; on site modules, we can place local settings, files, custom logic, and resources.



The distinction between public reusable modules and site modules is purely formal; they are both Puppet modules with standard structure. It may anyway make sense to place in separate directories (module paths) the ones you develop internally or modify from a public source from the public ones you use unaltered.

Let's see how these components may fit our Puppet tasks.

Definition of the classes to include in each node

This is typically done when, in Puppet, we talk about **nodes' classification**: the task that the Puppet Master accomplishes when it receives a client's request and determines the classes and parameters to use when compiling the relevant catalog.

Nodes' classification can be done in different ways:

- We can use the `node` object on `site.pp` and other manifests eventually imported from there. In this way we identify each node by `certname` and declare all the resources and classes we want for it:

```
node 'web01.example.com' {  
    include ::general  
    include ::apache  
}
```
- Here we may even decide to follow a **node-less** layout, where we don't use the node object at all and rely on facts to manage what classes and parameters to assign to our nodes. An example of this approach is examined later.
- On an **ENC** where can be defined the classes (and parameters) that each node should have. The returned YAML for our simple case would be something like the following:

```
---  
classes:  
  - general:  
  - apache:  
parameters:  
  dns_servers:  
    - 8.8.8.8  
    - 8.8.4.4  
  smtp_server: smtp.example.com  
environment: production
```

- Via LDAP, where we can have a hierarchical structure where a node can inherit the classes (referenced with the `puppetClass` attribute) set in a parent node (`parentNode`).
- On Hiera, using the `hiera_include` function; just add in `site.pp`:
`hiera_include('classes')`



We then define in our hierarchy, under the key `classes`, what to include for each node. For example, with a YAML backend, our case would be represented with the following:

```
---
classes:
  - general
  - apache
```

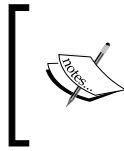
- On **site module(s)** any custom logic can be placed as, for example, the classes and resources to include for all the nodes or for specific groups of nodes.

Definition of the parameters to use for each node

This is another crucial part, as with parameters we can characterize our nodes and define the resources we want for them.

Generally, to **identify** and characterize a node in order to differentiate it from the others and provide the specific resources we want for it, we need very few key parameters, such as (names used here may be common, but are arbitrary and are not Puppet's internal ones):

- **role**, is almost a standard de facto name to identify the kind of server; a node is supposed to have just one role, which might be something like `webserver`, `app_be`, `db` or anything that identifies the function of the node. Note that web servers that serve different web applications should have different roles (that is `webserver_site`, `webserver_blog`). We can have one or more nodes with the same role.
- **env**, or any name that identifies the operational environment of the node (is it a development, test, qa, or production server?).



Note that this doesn't necessarily match Puppet's internal `environment` variable. Some prefer to merge the `env` information inside the `role`, having roles like `webserver_prod`, or `webserver_devel`.

- **zone, site, datacenter, country**, or any parameter that might identify the network, country, availability zone, or datacenter where the node is placed. A node is supposed to belong to only one of these. We might not require this in our infrastructure.
- **tenant, component, application, project**, or cluster might be other kinds of variables that characterize our node. There's no real standard for their naming, and their usage and necessity strictly depend on the underlying infrastructure.

With parameters like these, any node can be fully identified and be served with any specific configuration. It makes sense to provide them, where possible, as facts.

The parameters and the variables we use in our manifests may have different natures, such as:

- Role/env/zone as defined before, to identify the nodes; they are typically used to determine the values of other parameters
- OS related parameters, like package names and file paths
- Variables that define services of our infrastructure (DNS servers, NTP servers...)
- Usernames and passwords, which should be reserved, used to manage credentials
- Parameters that express any further custom logic and classifying need (master, slave, host_number...)
- Parameters exposed by the parameterized classes or defines we use

Many times, the values of these variables and parameters have to change according to the values of other variables, and it's important to have a general idea, from the beginning, of what the variations involved and the possible exceptions are, as we will probably define our logic according to them. As a general rule we will most of the time use the **identifying parameters** (role/env/zone...) to define most of the other parameters, so we'll probably need to use them in our Hiera hierarchy or in Puppet selectors. This also means that we will probably need to set them as **top scope variables** (for example via an ENC) or **facts**.

As with classes to include, parameters may be set by various components; some of them are actually the same, since in Puppet, node classification involves both classes to include and parameters to apply:

- On `site.pp`, we can set variables. If they are outside nodes' definitions they are at top scope, and if they are inside they are at node scope. Top scope variables should be referenced with a `::` prefix, for example `::role`. Node scope variables are available inside the node's classes with their plain name: `$role`.
- An ENC returns parameters, treated as top scope variables, alongside classes, and the logic of how they can be set depends entirely on their structure. Popular ENCs such as the Foreman, Puppet Dashboard, and Puppet Enterprise allow users to set variables for single nodes or for groups, often in a hierarchical fashion. The kind and amount of parameters set here depend on how much information we want to manage on the ENC and how much to manage somewhere else.
- LDAP, when used as node classifier, returns variables for each node as defined with the `puppetVar` attribute. They are all set at top scope.



On Hiera we set keys which we can map to Puppet variables with the functions `hiera()`, `hiera_array()`, and `hiera_hash()` inside our Puppet code. Since version 3, Puppet's **data bindings** automatically look up class parameters from Hiera data, mapping parameter names to Hiera keys, so for these cases we don't have to explicitly use `hiera*` functions. The defined hierarchy determines how the keys' values change according to the values of other variables. On Hiera, ideally, we should place variables related to our infrastructure and credentials, but not OS related variables (they should stay in modules if we want them to be reusable).

A lot of documentation about Hiera shows sample hierarchies with facts like `osfamily` and `operatingsystem`. In my very personal opinion, such variables should not stay there (weighting the hierarchy size), since OS differences should be managed in the classes and modules used and not in Hiera. Specific parameters for a deployment should be in data; common things that may vary between operating systems should be in the module implementation.

- On **shared modules**, we typically deal with OS specific parameters. Modules should be considered as reusable components that know all about how to manage the homonymous application on different OS but nothing about custom logic: they should expose parameters and defines that allow users to determine their behavior and fit their own needs.

- On **site module(s)**, we can place infrastructural parameters or any custom logic more or less based on other variables.
- Finally, it's possible, and generally recommended, to create custom facts that identify the node directly from the client. A case of this approach is a totally facts-driven infrastructure, where all the nodes identifying variables, upon which all the other parameters are defined, are set as facts.

Definition of the configuration files provided to the nodes

It's almost certain that we will need to manage configuration files with Puppet and that we need to store them somewhere, either as plain static files to serve via Puppet's `fileserver` functionality using the `source` argument of the `File` type, or via `.erb` templates.

While it's possible to configure custom `fileserver` *shares* for static files and absolute paths for templates, it's definitely recommended to rely on the modules' auto-loading conventions and place such files inside custom or shared modules, unless we decide to use Hierarchical for them.

Configuration files, therefore, are typically placed in the following:

- **Shared modules:** These may provide default templates that use variables exposed as parameters by the modules' classes and defines them. As users, we don't directly manage the module's template but the variables used inside it. A good and reusable module should allow us to override the default template with a custom one. In this case, our custom template should be placed in a site module. If we've forked a public shared module and maintain a custom version, we might be tempted to place all our custom files and templates there. In doing so, we lose in reusability and gain, maybe, in short term usage simplicity.
- **Site module(s):** These are, instead, a more correct place for custom files and templates if we want to maintain a setup based on public shared modules, which are not forked, and custom site ones where all our stuff stays confined in a single or few modules. This allows us to recreate similar setups just by copying and modifying our site modules, as all our logic, files, and resources are concentrated there.
- **Hiera:** Thanks to the smart `hiera-file` backend, this can be an interesting alternative place to store configuration files, both static ones and templates. We can benefit from the hierarchy logic that works for us and can manage any kind of file without touching modules.

- Custom **fileserver** mounts can be used to serve any kind of static files from any directory of the Puppet Master. They can be useful if we need to provide via Puppet files generated/managed by third party scripts or tools. An entry in `/etc/puppet/fileserver.conf` is like the following:

```
[data]
path /etc/puppet/static_files
allow *
```

- Allows us to serve a file such as `/etc/puppet/static_files/generated/file.txt` with the argument:
`source => 'puppet:///data/generated/file.txt',`

Definition of custom resources and classes

We'll probably need to provide custom resources to our nodes that are not declared in the shared modules because they are too specific, and we'll probably want to create some grouping classes, for example, to manage the common baseline of resources and classes we want applied to all our nodes.

This is typically a bunch of custom code and logic that we have to place somewhere. The usual locations are as follows:

- **Shared modules:** These are forked and modified to include custom resources; as already outlined, this approach doesn't pay in the long term.
- **Site module(s):** The preferred place-to-place custom stuff, including some classes where we can manage common baselines, role classes and other container classes.
- **Hiera:** Partially, if we are fond of the `create_resources` function fed by hashes provided in Hiera. In this case, in a site, shared module, or maybe even in `site.pp`, we have to place the `create_resources` statements somewhere.

The Foreman

The Foreman is definitively the biggest open source software related to Puppet, and not directly developed by Puppet Labs.

The project was started by Ohad Levy, who now works at Red Hat and leads its development, supported by a great team of internal employees and community members.

The Foreman can work as a Puppet ENC and reporting tool, it presents an alternative to the inventory system, and most of all, it can manage the whole lifecycle of the system, from provisioning, to configuration and dismissal.

Some of its features have been quite ahead of their time.

For example, the `foreman()` function made possible what is now done with the `puppetdbquery` module.

It allows direct query of all the data gathered by the Foreman: facts, nodes classification, and Puppet run reports.

Let's look at this example, which assigns to the variable `$web_servers` the list of hosts that belong to the `web` hostgroup that have reported successfully in the last hour:

```
$web_servers = foreman("hosts", "hostgroup ~ web and status.failed = 0
and last_report < \"1 hour ago\"")
```

This was possible before PuppetDB was even conceived.

The Foreman really deserves at least one book by itself, so here we will just summarize its features and explore how it can fit in to a Puppet architecture.

We can decide which of the following components to use:

- Systems provisioning and life-cycle management
- Nodes IP addressing and naming
- The Puppet ENC function, based on a complete web interface
- Management of client certificates on the Puppet Master
- The Puppet reporting function, with a powerful query interface
- The facts querying function, equivalent to Puppet's inventory system

For some of these features, we may need to install Foreman's **Smart Proxies** on some infrastructural servers. The proxies are registered on the central Foreman server and provide a way to remotely control relevant services (DHCP, PXE, DNS, Puppet Master, and so on).

The web GUI, based on Rails, is complete and appealing, but it might turn out to be cumbersome when we have to deal with a large number of nodes; for this reason, we can also manage the Foreman via CLI.



The original `foreman-cli` command has been around for years but is now deprecated for the new `hammer` (<https://github.com/theforeman/hammer-cli>) with the Foreman plugin, which is very versatile and powerful as it allows to manage, via the command line, most of what we can do on the web interface.

Roles and profiles

In 2012, Craig Dunn wrote a blog post (<http://www.craigdunn.org/2012/05/239/>), which quickly became a point of reference on how to organize Puppet code. He discussed **roles** and **profiles**. The role describes what the server represents: a live web server, a development web server, a mail server, and so on. Each node can have one, and only one, role. Note that in his post he manages environments inside roles (two web servers on two different environments have two different roles), as follows:

```
node www1 {
  include ::role::www::dev
}
node www2 {
  include ::role::www::live
}
node smtp1 {
  include ::role::mailserver
}
```

Then he introduces the concept of profiles, which include and manage modules to define a logical technical stack. A role can include one or more profiles:

```
class role {
  include profile::base
}
class role::www inherits role {
  include ::profile::tomcat
}
```

In environment related sub roles, we can manage the exceptions we need (here, for example, the `www::dev` role includes both the `database` and the `webserver::dev` profile):

```
class role::www::dev inherits role::www {
  include ::profile::webserver::dev
  include ::profile::database
}
```

```
class role::www::live inherits role::www {
  include ::profile::webserver::live
}
```

Inheritance is usually discouraged in Puppet, particularly on parameterized classes, and it was even removed from nodes. Usage of class inheritance here is not mandatory, but it results in minimized code duplication and doesn't have the problems of parameterized classes, as these classes are only used to group other classes. Alternatives to inheritance could be to include a base profile instead of having a base class, or to consider all roles independent between them and duplicate their contents. The best option would depend on the kind of roles have in our deployment.

This model expects modules to be the only components where resources are actually defined and managed; they are supposed to be reusable (we use them without modifying them) and manage only the components they are written for.

In profiles, we can manage resource and class ordering, we can initialize variables and use them as values for arguments in the declared classes, and we can generally benefit from having an extra layer of abstraction:

```
class profile::base {
  include ::networking
  include ::users
}
class profile::tomcat {
  class { 'jdk': }
  class { 'tomcat': }
}
class profile::webserver {
  class { 'httpd': }
  class { 'php': }
  class { 'memcache': }
}
```

In profiles subclasses we can manage exceptions or particular cases:

```
class profile::webserver::dev inherits profile::webserver {
  Class['php'] {
    loglevel => "debug"
  }
}
```

This model is quite flexible and has gained a lot of attention and endorsement from Puppet Labs. It's not the only approach that we can follow to organize, in the same way, the resources we need for our nodes, but it's a current best practice and a good point of reference, since it formalizes the concept of role and exposes how we can organize and add layers of abstraction between our nodes and the used modules well.



Note that given the above naming, we would have custom (site) `role` and `profile` modules, where we place all our logic. Placing these classes inside a single site module (for example: `site::role::www`) is absolutely the same and basically just a matter of personal taste and naming and directories layout.

The data and the code

Hiera's main reason for existence is data separation. In practical terms, this means that we can simplify code that includes the selection of the data to use in the very code, as in this example:

```
$dns_server = $zone ? {
  'it'      => '1.2.3.4',
  default => '8.8.8.8',
}
class { '::resolver':
  server => $dns_servers,
}
```

It can be converted into something where there's no trace of local settings, as in the following:

```
$dns_server = hiera('dns_server')
class { '::resolver':
  server => $dns_servers,
}
```

Since version 3 of Puppet, the above code can be simplified even more by just including the `::resolver` class and Hiera will look up in its data files the value to resolve the `resolver::server` key.

The advantages of having data (in this case the IP of the DNS server, whatever is the logic to elaborate it) in a separate place are as follows:

- We can manage and modify data without changing our code
- Different people can work on data and on code

- Hiera pluggable backend system dramatically enhances how and where data can be managed, allowing seamless integration with third party tools and data sources
- Code layout is simpler and more error-proof
- The lookup hierarchy is configurable

Nevertheless, there are a few drawbacks, or maybe just necessary side effects or *needed evolutionary steps*, with it, which are as follows:

- What we learnt about Puppet and used to do without Hiera is obsolete
- We don't see the values we are using directly in our code
- We have two different places to look at to understand what code does
- We need to have set the variables we use in our hierarchy as top scope variables or facts, or at least be able to refer them with a fixed, fully qualified name
- We might have to refactor a lot of existing code to move our data and logic into Hiera



A personal note: I've been quite a late jumper on the Hiera wagon; while developing modules with the ambition to make them reusable, I decided I couldn't exclude users that weren't using this additional component and so, until Puppet 3 with Hiera integrated became mainstream, I didn't want to force the usage of Hiera in my code.

Now things are different: Puppet 3's data bindings change the whole scene, Hiera is deeply integrated and is here to stay, and so, even if we can happily live without using it, I would definitely recommend its usage in most cases.

Sample architectures

We have outlined the main tasks and components we can use to put things together in a Puppet architecture; we have given a look at Foreman, Hiera, and at the roles and profiles pattern; now let's see some real examples based on them.

The default approach

By default, Puppet doesn't use an ENC and lets us classify nodes directly in `/etc/puppet/manifests/site.pp` (or in files imported from there) with the `node` statement. So a very basic setup would have `site.pp` with content like the following:

```
node www01 {
  # Place here resources to apply to this node in Puppet DSL:
  # file, package service, mount...
}
node lb01 {
  # Resources for this node: file, package service...
}
```

This is all we need: no modules with their classes, no Hiera, no ENC; just good old plain Puppet code as they teach us in schools, so to speak.

This basic approach, useful just for the first tests, obviously does not scale well and would quickly become a huge mess of duplicated code.

The next step is to use classes which group resources, and if these classes are placed inside modules, we can include them directly without the need to explicitly import the containing files:

```
node www01 {
  include ::apache
  include ::php
}
```

Also, this approach, even if definitely cleaner, will quickly be overwhelmed by redundant code, and so we will probably want to introduce **grouping classes** that group other classes and resources according to the desired logic.

One common example is a class that includes all the modules, classes and resources we want to apply to all our nodes: a `general` class.

Another example is `role` classes, which include all the extra resources needed by a particular node:

```
node www01 {
  include ::general
  include ::role::www
}
```

We can then have other grouping classes to better organize and reuse our resources, such as the profiles we have just discussed.



Note that with the above names we would need two different local (site) modules: `general` and `role`; I personally prefer to place all the local, custom resources in a single module, to be called `site` or, even better, with the name of the project, customer, or company. Given this, the previous example could be as follows:

```
node www01 {  
  include ::site  
  include ::site::role::www  
}
```

These are only naming matters, which have consequences on directories layout and eventually on permissions management on our SCM, but the principle of grouping resources according to custom logic is the same.

Up to now, we have just included classes and often the same classes are included by nodes that need different effects from them, for example, slightly different configuration files or specific resources, or in case of any kind of variation we have in the real world while configuring the same application on different systems.

Here is where we need to use variables and parameters to alter the behavior of a class according to custom needs.

And here is where the complexity begins, because there are various elements to consider, such as the following:

- Which variables identify our node
- If they are sufficient to manage all the variations in our nodes
- Where we want to place our logic that copes with them
- Where configurations should be provided as plain static files, where it is better to use templates, or where we could just modify single lines inside files
- How these choices may affect the risk of making a change that affects unexpected nodes



The most frequent and dangerous mistakes with Puppet are due to people making changes in code (or data) that are supposed to be made for a specific node but also affect other nodes. Most of the time this happens when people don't know the structure and logic of the Puppet codebase they are working on well enough. There are no easy rules to prevent such problems, just some general suggestions, such as the following:

- Promote code peer review and communication among the Puppeteers
- Test code changes on canary nodes
- Use naming conventions and coherent code organization to maximize the principle of least surprise
- Embrace code simplicity, readability and documentation
- Be wary of the scope and extent of our abstraction layers

We also need classes that actually allow things to be changed, via parameters or variables, if we want to avoid placing our logic directly inside them.

Patterns on how to manage variables and their effect on the applied resources have changed a lot with the evolution of Puppet and the introduction of new tools and functionalities.

We won't indulge in how things were done in the *good old days*. In a modern, and currently recommended Puppet setup, we expect to have the following:

- At least Puppet 3 on the Puppet Master to eventually enjoy data bindings
- Classes that expose parameters that allow us to manage them
- Reusable public modules that allow us to adapt them to our use case without modifications

In this case, we can basically follow two different approaches:

We can keep on including classes and set on Hiera the values we want for the parameters we need to modify. So, in our example we could have in `site/manifests/role/www.pp` something like the following:

```
class site::role::www {
  include ::apache
  include ::php
}
```


The same is true on Hiera for a file like `hieradata/env/devel.yaml`, where we set parameters like the following:

```
---
apache::port: 8080
```

Alternatively, we might use explicit class declarations such as:

```
class site::role::www {
  $apache_port = $env ? {
    devel    => '8080',
    default => '80',
  }
  class { ['::apache']:
    port => $apache_port,
  }
  include ::php
}
```

Data, and logic on how to determine it, is definitively inside code.

Basic ENC, logic on site module, Hiera backend

The ENC and Hiera can be alternative or complementary; this approach gets advantages from both and uses the site module for most of the logic for class inclusion, the configuration files, and the custom resources.

In Hiera, all the class parameters are placed.

In the ENC, when not possible via facts, we set the variables that identify our nodes, and can be used on our Hiera hierarchy.

In `site.pp` in the same ENC, we include just a single site class and here we manage our grouping logic. For example, with a general baseline and role classes:

```
class site {
  include ::site::general
  if $::role {
    include "::site::roles::${::role}"
  }
}
```

In our role classes, which are included if the `$role` variable is set on the ENC, we can manage all the role-specific resources, eventually dealing with differences according to environment or other identifying variables directly in the role class, or using profiles.



Note that in this chapter we've always referred to class names with their full name, so a class such as `mysql` is referred with `::mysql`. This is useful for avoiding name collisions when, for example, role names may clash with existing modules. If we don't use the leading `::` chars, we will have problems, for example, with a class called `site::role::mysql`, which may mess with the main class `mysql`.

The Foreman and Hiera

The Foreman can act as a Puppet ENC; it's probably the most common ENC around and we can use both Foreman and Hiera in our architecture.

In this case we should strictly separate responsibilities and scopes, even if they might be overlapping. Let's review our components and how they might fit in a scenario based on the Foreman, Hiera, and the usual site module(s):

- **Classes** to include in nodes: This can be done on the Foreman, the site module, or both. It mostly depends on how much logic we want on the Foreman, and so how many activities have to be done via its interface and how many are moved into site module(s). We can decide to define roles and profiles on the site module and use Foreman just to define top scope variables and the inclusion of a single basic class, as in the previous example. Alternatively, we may prefer to use Foreman's HostGroups to classify and group nodes, moving into Foreman most of the classes grouping logic.
- **Variables** to assign to nodes: This can be done on Foreman and Hiera. It probably makes sense to set on Foreman only the variables we need to identify nodes (if they are not provided by facts) and generally the ones we might need to use on the Hiera's hierarchy. All the other variables and the logic on how to derive them should stay on Hiera.
- **Files** should stay on our site module or, eventually, on Hiera (with the `hiera-file` plugin).

Hiera-based setup

A common scenario involves the usage of Hiera to manage both the classes to include in nodes and their parameters.

No ENC is used; `site.pp` just needs the following, together, eventually, with a few handy resource defaults:

```
hiera_include('classes')
```

Classes and parameters can be assigned to nodes enjoying the flexibility of our hierarchy, so in a `common.yaml` we can have the following:

```
---
# Common classes on all nodes
classes:
  - puppet
  - openssh
  - timezone
  - resolver
# Common Class Settings
timezone::timezone: 'Europe/Rome'
resolver::dns_servers:
  - 8.8.8.8
  - 8.8.4.4
```

In a specific data source file such as `role/web.yaml`, add the classes and the parameters we want to apply to that group of nodes:

```
---
classes:
  - stack_lamp
stack_lamp::apache_include: true
stack_lamp::php_include: true
stack_lamp::mysql_include: false
```

The modules used (here a sample `stack_lamp`, but it could be something such as `profile::webserver` or `apache` and `php`) should expose parameters that are needed to configure things as expected.

They should also allow creation of custom resources, such as `apache::vhost`, by providing hashes to feed a `create_resources()` function present in a used class.

Configuration files and templates can be placed in a site module with, eventually, additional custom classes.

We can also use the `hiera-file` plugin to deliver configuration files having a Hierarchical-only setup. This is a somewhat extreme approach. Everything is managed by Hierarchical: the classes to include in nodes, their parameters, and also the configuration files to serve to clients. Also, here we need modules and relevant classes that expose parameters to manage the content of files.

Secrets, credentials, and sensitive data may be encrypted via `hiera-eyaml` or `hiera-gpg`.

We may wonder if a site module is still needed, since most of its common functions (providing custom files, managing logic, defining and managing variables) can be moved to Hierarchical.

The answer is probably yes; even in a similar, strongly Hierarchical-oriented scenario, a site module is probably needed. We might for example use custom classes to manage edge cases or exceptions that could be difficult to replicate with Hierarchical without adding a specific entry in the hierarchy.

One important point to consider when we move most of our logic to Hierarchical is how much this costs in terms of hierarchy size. Sometimes a simple (even if not elegant) custom class that deals with a particular case may save us from adding a layer in the hierarchy.

Foreman smart variables

This is the Foreman alternative approach to Hierarchical for the full management of the variables used by nodes.

Foreman can automatically detect the parameters exposed by classes and allows us to set values for them according to custom logic, providing them as parameters for parameterized classes via the ENC functionality (support for parameterized classes via ENC has been available since Puppet 2.6.5).

To each class we can map one or more **smart variables**, which may have different values according to different, customizable conditions and hierarchies.

The logic is somewhat similar to Hierarchical, with the notable difference that we can have a different hierarchy for each variable and have other ways to define its content via matchers.

User experience benefits from web interface and may turn out to be easier than editing Hierarchical files directly. Foreman auditing features allow us to track changes as would a SCM on plain files.

We don't have the multiple backend flexibility that we have on Hieria and we'll be completely tied to Foreman for the management on our nodes.

Personally, I have no idea how many people are extensively using smart variables in their setups; just be aware that there exists this alternative for data management.

Facts-driven truths

A facts driven approach was theorized by Jordan Sissel, Logstash's author in a 2010 blog post (<http://www.semicomplete.com/blog/geekery/puppet-nodeless-configuration>). The most authoritative information we can have about a node comes from its own facts.

We may decide to use facts in various places: in our hierarchy, in our site code, in templates, and if our facts permit us to identify the node's role, environment, zone, or any identifying variable, we might not even need node classification and manage all our stuff in our site module or on Hieria.

It is now very easy to add custom facts placing a file in the node's `/etc/facter/facts.d` directory. This can be done, for instance, by a (cloud) provisioning script.

Alternatively, if our nodes' names are standardized and informative, we can easily define our identifying variables in facts that might be provided by our site module.

If all the variables that identify our node come from facts, we can have in our `site.pp` a single line as simple as the following:

```
include site
```

In our `site/manifests/init.pp` have something like:

```
class site {
  if $::role {
    include "site::roles::role_${::role}"
  }
}
```

The top scope `$::role` variable would be, obviously, a fact.

Logic for data and classes to include can be managed where we prefer: on site modules, Hieria, or the ENC.

The principle here is that as much data as possible, and especially the nodes' identifying variables, should come from facts.

Also, in this case common sense applies and extreme usage deviations should be avoided; in particular, a custom ruby fact should compute its output without any local data. If we start to place data inside the fact in order to return data, we are probably doing something wrong.

Nodeless site.pp

We have seen that `site.pp` does not necessarily need to have node definitions in its content in imported files. We don't need them when we drive everything via facts, where we manage class inclusion in Hiera, and we don't need them with an approach where conditionals based on host names are used to set the top scope variables that identify our nodes:

```
# nodeless site.pp

# Roles are based on hostnames
case $::hostname {
  /^web/: { $role = 'web' }
  /^puppet$/: { $role = 'puppet' }
  /^lb/: { $role = 'lb' }
  /^log/: { $role = 'log' }
  /^db/: { $role = 'db' }
  /^el/: { $role = 'el' }
  /^monitor/: { $role = 'monitor' }
  default: { $role = 'default' }
}

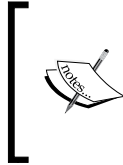
# Env is based on hostname or (sub) domain
if 'devel' in $::fqdn { $env = 'devel' }
elsif 'test' in $::fqdn { $env = 'test' }
elsif 'qa' in $::fqdn { $env = 'qa' }
else { $env = 'prod' }

include site
# hiera_include('classes')
```

Here, the `$role` and `$env` variables are set at top scope according to hostnames that would benefit from a naming standard we can parse with Puppet code.

At the end, we just include our `site` class or use `hiera_include` to manage the grouping logic for what classes to include in our nodes.

Such an approach makes sense only where we don't have to manage many different hostnames or roles, and where the names of our nodes follow a naming pattern that lets us derive identifying variables.



Note that the `$::hostname` or `$clientcert` variables might be forged and may return non-trustable values. Since Puppet 3.4, if in `puppet.conf` we set `trusted_node_data = true`, we have at our disposal the special variable `$trusted['certname']` to identify a verified hostname.

Summary

We have examined the tasks we have to deal with: how to manage nodes, the logic we group them with, the parameters of the classes we use, and the configuration files. We have reviewed the tools at our disposal: an ENC like the Foreman, Hiera, and our custom site modules.

We have seen some samples of how these elements can be managed in different combinations. Now it is time to explore more deeply an important component of a Puppet setup – modules – and see how to write really reusable ones.

5

Using and Writing Reusable Modules

People in the Puppet community have always wondered how to write code that could be reused. Earlier, this was done with **recipes**, collected on the old wiki, where people shared fragments of code for specific tasks. Then we were introduced to **modules**, which allowed users to present all the Puppet and Ruby code and configuration files needed to manage a specific application in a unique directory.

People started writing modules, someone even made a full collection of them (the father of all the modules collections is David Schmitt; then others followed), and, at the European Puppet Camp in 2010, Luke Kanies announced the launch of the **Puppet Modules Forge**, a central repository of modules which can be installed and managed directly from the command line.

It seemed the solution to the already growing mess of unstructured, sparse, interoperable, and incompatible modules, but, in reality, it took some time before becoming the powerful resource it is now.

In this chapter, we will review the following:

- The evolution of modules layout
- The parameters dilemma: what class parameters have to be exposed and where
- Principles for modules' reusability

Modules layout evolution

Over the years, different modules layouts have been explored, following the evolution of Puppet's features and the refinement of usage patterns.

There has never been a unique way of doing a module, but patterns and best practices have emerged and we are going to review the most relevant ones.

Class parameters—from zero to data bindings

The introduction of parameterized classes, with Puppet 2.6, has been a crucial step in standardizing the interfaces of classes. On earlier versions there was no unique way to pass data to a class. Variables defined anywhere could be dynamically used inside Puppet code or in templates to manage the module's behavior; there was no standard API to access or set them. We used to define parameter less classes as follows:

```
class apache {  
  # Variables used in DSL or in templates were dynamically scoped  
  # and referenced without using their fully qualified name.  
  # IE: $port, not $apache::port or $::apache_port  
}
```

To declare them always and only with Apache as follows:

```
include apache
```

The introduction of parameters in classes has been important because it allowed a single entry point for class data:

```
class apache (  
  $port = 80 ) {  
}
```

The default value of the parameter could be overridden with an explicit declaration, such as the following:

```
class { 'apache':  
  port => 8080,  
}
```

Such a solution, anyway, has not been completely decisive: usage of parameterized classes introduced new challenges, such as the need to declare them only once in our catalog for each node. This forced people to rethink some of their assumptions on how and where to make class inclusion in their code.

We still could `include apache` as many times as we wanted in a catalog but we didn't have any method to set specific parameters if the class didn't explicitly manage a way to lookup for external variables, for example with a syntax like the following:

```
class apache (
  $port = hiera('apache::port', '80') {
  }
```

This, obviously, would have required all the module's users to use Hiera.

I wouldn't dare to say that the circle has been closed in Puppet 3's data bindings: the automatic Hiera lookup of class parameters allows setting parameters via both explicit declaration and plain inclusion with parameter values set in Hiera.

After years of pain, alternative solutions, creative and unorthodox approaches and evolution of the tool, I'd say that now the mainstream and recommended way to use classes is to just include them and manage their parameters on Hiera, using Puppet 3's data bindings feature.

On the manifests, we can declare classes with the following:

```
include apache
```

Be sure that whatever parse order is followed by Puppet, its data can be defined in Hiera files, so with a YAML backend, we'll use a syntax as simple as the following:

```
---
apache::port: '8080'
```

Params pattern

When people had to cope with different OS in a module, they typically started using **selectors** or **conditionals** for assigning variables or parameters the correct values according to facts such as `operatingsystem` and `operatingsystemrelease`, and the more recent `osfamily`.

A typical case with a **selector** would be as follows:

```
class apache {
  $apache_name = $::operatingsystem ? {
    /(?:Debian|Ubuntu|Mint)/    => 'apache2',
    /(?:RedHat|CentOS|Scientific)/ => 'httpd',
    default                     => 'apache'
  }
  package { $apache_name:
    ensure => present,
  }
}
```

Having this mix of variable definitions and resource declarations was far from elegant and in some time people started to place in a dedicated class, usually called `params`, the management of the module's variables.

They can be set with selectors, as in the previous example, or, more commonly, inside `case` statements, always based on facts related to the underlying OS:

```
class apache::params {
  case $::osfamily {
    'RedHat': {
      $apache_name = 'httpd'
    }
    'Debian': {
      $apache_name = 'apache2'
    }
    default: {
      fail("Operating system ${::operatingsystem} not supported")
    }
  }
}
```

The main class has to just include the `params` class and refer to internal variables using their fully qualified name:

```
class apache {
  include apache::params
  package { $apache::params::apache_name:
    ensure => present,
  }
}
```

This is a basic implementation of the so-called `params` pattern, and has the advantage of having a single place where we define all the internal variables of a module or the default values for its parameters.

In the next example, the package name is also exposed as a parameter (this can be considered a reusability feature, as it allows users to override the default package name for the application that is going to be installed), and since the default value is defined in `params.pp`, the main class has to inherit it:

```
class apache (
  $package_name = $apache::params::package_name,
) inherits apache::params {
  package { $apache::params::package_name:
    ensure => present,
  }
}
```

The `params` pattern has been widely used and it works well. Still, it embraces a code and data mixup that so many wanted to avoid.

Data in modules

The first proposals about the way to separate modules' data from their code date back to 2010, with a blog post from Dan Bode titled *Proposal: Managing Puppet's configuration data*.

At that time, Hiera was still not available (the post refers to its ancestor, `extlookup`) but most of the principles described there have been considered when a solution was implemented.



Dan Bode's blog was closed, but the article is still available in the Internet archive <https://web.archive.org/web/20121027061105/http://bodepd.com/wordpress/?p=64>

When Hiera was introduced, it seemed a good solution to manage OS related variations via its hierarchy. It soon became clear, anyway, that global site related data, as it can be the one we place on our data sources, is not an appropriate backend for modules' internal data if we want them to be reusable and distributable.

Possible solutions, inspired or derived from Dan's post, have been identified for some time, but only with the release of Puppet 3.3.0 was it converted into reality as an experimental feature that finally addressed what's generally summarized by the term **Data in modules**: have a dedicated hierarchy inside the module and its relevant Hiera data.

It seemed finally, the long sought after solution to have data separation also for modules internal data, but it failed to pass Puppet Labs' user acceptance tests.

It was not so easy for modules authors to manage and this implementation has been removed in the following Puppet versions, but the issue is too important to be ignored, so R.I.Pienaar proposed an implementation based on an independent module: <https://github.com/ripienaar/puppet-module-data>

This approach is much simpler to use, doesn't require big changes to existing code; being implemented as a module, it can be used on most Puppet installations (version 3.x is required), and does exactly what we expect.

Files and class names

Besides the `init.pp` file, with the main class, all the other classes defined in the manifests directory of a module can have any name. Still, some names are more common than others. We have seen that `params.pp` is a sort of standard de facto pattern, and is not the only one. It's common, for example, to have files like `server.pp` and `client.pp` with subclasses to manage the `server/client` components of an application.

R.I.Pienaar (definitively one of the most influential contributors to Puppet's evolution) suggested in a blog post a module layout that involves splitting the main modules' resources in three different classes and relevant files: `install.pp` to manage the installation of the application, `config.pp` to manage its configuration, and `service.pp` to manage its service(s). So a typical `package-service-configuration` module can have in its `init.pp` file something like the following:

```
class postgresql [...] { [...]
  class{'postgresql::install': } ->
  class{'postgresql::config': } ~>
  class{'postgresql::service': }
}
```

And in the referred sub classes the relevant resources to manage.

This pattern has pros and cons. Its advantages are as follows:

- Clear separation of the components provided by the module
- Easier to manage relationships and dependencies, based not on single resources, which may vary, but on whole subclasses, which are always the same
- More compact and easier to read `init.pp`
- Naming standardization for common components

Some drawbacks can be as follows:

- Various extra objects are added to the catalog to do the same things: this might have performance implications at scale, even if the reduced number of relationships might balance the final count
- It is more cumbersome to manage the relationship logic via users' parameters (for example when we want to provide a parameter that defines whether to restart or not a service after a change on the configurations)
- For simple `package-service-config` modules, it looks redundant to have three extra classes with just a resource for each

In any case, be aware that such an approach requires the `contain` function (available from Puppet 3.4.0) or the usage of the `anchor` pattern to work correctly.

The anchor pattern

Puppet has had a long-standing issue that affected and confused many users for years: <https://tickets.puppetlabs.com/browse/PUP-99>. One of its effects is that when we define a dependency on a class, Puppet extends that dependency to the resources declared in that class, as we may expect, but not to other classes eventually declared (included) there.

This may create problems and lead to unexpected behaviors (dependencies not managed in the order expected) when referring to a class like the `postgresql` we have seen, where other sub classes are declared.

A widely used work around is the `anchor` pattern, defined by Puppet Labs' Jeff McCune.

It is based on the `anchor` type, included in Puppet Labs' `stdlib` module, which can be declared as a normal resource:

```
anchor { 'postgresql::start': }
anchor { 'postgresql::end': }
```

This can then be used to contain the declared classes in a dependency chain:

```
anchor { 'postgresql::start': } ->
class { 'postgresql::install': } ->
class { 'postgresql::config': } ~>
class { 'postgresql::service': } ->
anchor { 'postgresql::end': }
```

In this way, when we create a relationship that involves a whole class, like the following, we are sure that all the resources provided in the `postgresql` class are applied before the `wordpress` class once, because they are explicitly contained in the `anchor` resource type:

```
class { 'postgresql': } -> class { 'wordpress': }
```



The `stdlib` module provides general purpose resources that extend the Puppet core language to help develop new modules. For this purpose it includes stages, facts, functions, types and providers.

The parameters dilemma

In modules, we typically set up applications: most of the time this is done by installing packages, configuring files and managing services.

We can write a module that makes exactly what we need for our working scenario or we can try to design it having in mind that people with different needs and infrastructures may use it.

These people might be us in the future, when we'll have to manage a Puppet setup for another project or cope with different kinds of servers, or manage unexpected or nonstandard requirements, and we might regret not having written our code in an abstract enough and reusable way.

Basically, parameters, being our API to the module's functionality, are needed for exactly this: to let modules behave in different ways and do different things according to the values provided by users.

Hypothetically, we could enforce inside our code exactly what we need and therefore do this without the need of having any parameter. Maybe our code could be simpler to use and read at the beginning, but this would be a technical debt we will have to pay, sooner or later.

So we need parameters, and we need them to allow users' customization of what the module does, in order to adapt it to different use cases.

Still it is not so obvious to define what parameters to use and where to place them, let's review some possible cases.

There might be different kinds of parameters, which can be as follows:

- Manage variables that provide the **main configuration settings** for an application (`ntp_server`, `munin_server`, and so on)
- Manage most or **all configuration settings** for an application (for a sample `ntp.conf` they would be `driftfile`, `statistics`, `server`, `restrict`, and so on)
- Manage **triggers** that define which components of the module should be used (`install_server`, `enable_passenger`, and so on)
- Manage the attributes of the resources declared inside a class or define (`config_file_source`, `package_name`, and so on)

- Manage the **behavior** of the module or some of its components (`service_autorestart`, `debug`, `ensure`, and so on)
- Manage **external** parameters related to **applications**, references or classes (`db_server`, `db_user`, and so on) not directly managed by the module, but needed to configure its application

There are also various **places** where these parameters can be exposed and can act as **entry points** where to feed our configuration data:

- The **main** module's class, defined in the `init.pp` file
- Single **sub** classes which may act as single responsibility points for specific features (class names like `modulename::server` and `modulename::db::mysql`, with their own parameters that can be directly set)
- **Configuration classes**, used as entry points for various settings, to override modules' defaults (`modulename::settings`, `modulename::globals`)
- Normal **defines** provided by the module

There is not an established and common structure or agreement regarding the kind of parameters and where to expose them.

They depend upon several factors, but the main design decision that the module's author has to make, which heavily affects the kind of parameters to expose, is how configurations are provided: are they **file-based** or **setting-based**?

File-based configurations expect the module to manage the application configuration via files, generally provided as ERB templates whose content is derived from the parameters exposed by the classes or defines that use it.

Setting-based configurations have a more granular approach: the application configuration files are managed as single configuration entries, typically single lines, which compose the final file.

In the first case, where files are managed as a whole, the module's classes should:

- Expose parameters that define at least the application's main settings and, optionally, most or all the other settings, as long as they are managed in the default template
- Allow users to provide a custom template that overrides the module's default one, and can be used to manage custom settings which are not manageable via parameters
- Allow users to provide configurations as plain static files, to serve via the `source` argument, since some users may prefer to manage configuration files as is

In the second case, the module should do the following:

- Provide native or defined types that allow manipulation of single lines inside a configuration file
- Eventually provide classes (either a single main class or many subclasses) that expose as parameters all the possible configuration settings for the managed application (they might be many and hard to keep updated)



I don't have a clear answer on what would be the preferred approach; I suppose it depends on the kind of managed application, the user's preference, and the complexity and structure of the configuration files.

The good news is that a module's author can accommodate both the approaches and leave the choice to the user.

Naming standards

The modules ecosystem has grown erratically, with many people redoing modules for the same application, either forking existing ones or developing them from scratch.

In this case, quantity is not a plus: there's really no advantage for Puppet users to have dozens of different modules for a single application, written with different layouts, parameters names, entry points, OS coverage, and features sets.

Even worse, when we use modules that have dependencies on other modules (as described in `Modulefile`), we may quickly end up having conflicts that may prevent us from installing a module with the `puppet module` tool and force us to fork and fix, just for our case.

Module interoperability is a wider issue which has not yet been solved, but there is something that can be done if there is the common will: an attempt to **standardize naming conventions** for modules' class and parameters names.

The various benefits are as follows:

- Saner and simpler user experience
- Easier module interoperability
- Suggested reusability patterns
- Predictability in usage and development.

It's common sense that such an effort should be driven by Puppet Labs, but for the moment, it remains a community effort, under the label `stdmod`: <https://github.com/stdmod>.

The naming conventions defined there embrace some standard de facto naming and try to define general rules for parameters and class names. For the ones that affect the resource arguments, the pattern is `[prefix_]resource_attribute`, so these may be some parameters names:

```
config_file_path, config_file_content, package_name, init_config_file_path, client_package_name
```

Many other names are defined; the basic principle is that a `stdmod` compliant module is not expected to have them all, but if it exposes parameters that have the same functions, it should call them as defined by the standard.

Reusability patterns

Modules reusability is a topic that has got more and more attention in recent years, as the more people started use Puppet, more evident the need of having some common and shared code to manage common things.

Reusable modules' main characteristics are as follows:

- They can be used by different people without the need to modify their content
- They support different OS, and allow easy extension to new ones
- They allow users to override the default files provided by the module
- They might have an opinionated approach to the managed resources but don't force it
- They follow a single responsibility principle and should manage only the application they are made for

Reusability, we must underline, is not an all or nothing feature, we might have different levels of reusability to fulfill the needs of a varying percentage of users.

For example, a module might support Red Hat and Debian derivatives, but not Solaris or AIX: is it reusable? If we use the latter OS, definitely not; if we don't use them, yes, for us it is reusable.

I am personally a bit extreme about reusability, and according to me, a module should also do the following:

- Allow users to provide alternative classes for eventual dependencies from other modules, to ease interoperability
- Allow any kind of treatment of the managed configuration files, be that file or setting-based
- Allow alternative installation methods
- Allow users to provide their own classes for users or other resources which could be managed in custom and alternative ways
- Allow users to modify the default settings (calculated inside the module according to the underlining OS) for package and service names, file paths, and other more or less internal variables that are not always exposed as parameters
- Expose parameters that allow removal of the resources provided by the module (this is a functionality feature more than a reusability one)
- Abstract monitoring and firewalling features so that they are not directly tied to specific modules or applications

Managing files

Everything is a file in UNIX, and Puppet, most of the time, manages files.

A module can expose parameters that allow its users to manipulate configuration files and it can follow one or both the files/setting approaches, as they are not alternative and can coexist.

To manage the contents of a file, Puppet provides different alternative solutions:

- Use **templates**, populated with variables that come from parameters, facts or any scope (argument for the `File` type:

```
content => template('modulename/path/templatefile.erb')
```

- Use **static** files, served by the Puppet server:

```
source => 'puppet:///modules/modulename/file'
```

- Manage the file content via `concat` (<https://github.com/puppetlabs/puppetlabs-concat>) a module that provides resources that allow you to build a file joining different fragments:

```
concat { $motd:  
  owner => 'root',  
  group => 'root',
```

```

    mode => '0644',
  }

  concat::fragment{ 'motd_header':
    target => $motd,
    content => "\nNode configuration managed by Puppet\n\n",
  }

```

- Manage the file contents via **augeas**, a native type that interfaces with the **augeas** configuration editing tool that manages configuration files with a key-value model (<http://augeas.net/>):

```

augeas { "sshd_config":
  changes => [
    "set /files/etc/ssh/sshd_config/PermitRootLogin no",
  ],
}

```

- Manage it with alternative in-file line editing tools

For the first two cases, we can expose parameters which allow us to define the module's main configuration file, either directly via the `source` and `content` arguments, or by specifying the name of the template to be passed to the `template()` function:

```

class redis (
  $config_file           = $redis::params::file,
  $config_file_source    = undef,
  $config_file_template  = undef,
  $config_file_content   = undef,
) {

```

We can manage the configuration file arguments with the following:

```

$managed_config_file_content = $config_file_content ? {
  undef => $config_file_template ? {
    undef => undef,
    default => template($config_file_template),
  },
  default => $config_file_content,
}

```

The `$managed_config_file_content` variable computed here takes the value of the `$config_file_content`, if present; otherwise it uses the template defined with `$config_file_template`. If this parameter is unset, the value is `undef`.

```

if $redis::config_file {
  file { 'redis.conf':

```

```
    path    => $redis::config_file,
    source  => $redis::config_file_source,
    content => $redis::managed_config_file_content,
  }
}
```

In this way, users can populate `redis.conf`, either via a custom template (placed in the `site` module), as follows:

```
class { 'redis':
  config_file_template => 'site/redis/redis.conf.erb',
}
```

Or directly provide the content attribute, as shown here:

```
class { 'redis':
  config_file_content => template('site/redis/redis.conf.erb'),
}
```

Or, finally, provide a fileserver source path, such as the following:

```
class { 'redis':
  config_file_source => 'puppet:///modules/site/redis/redis.conf',
}
```

In case users prefer to manage the file in other ways (augeas, concat, and so on), they can just include the main class, which by default does not manage the configuration file contents, and use whichever method they prefer to alter them:

```
class { 'redis': }
```

A good module could also provide custom defines that allow easy and direct ways to alter configuration files' single lines, either using augeas or other in-file line management tools.

Managing configuration hash patterns

If we want a full **infrastructure as data** setup and to be able to manage as data all our configuration settings, we can follow two approaches, regarding the number, name, and kind of parameters to expose:

- Provide a parameter for each configuration entry we want to manage
- Provide a single parameter which expects a hash where any configuration entry may be defined

The first approach requires a substantial and ongoing effort, as we have to keep our module's classes updated with all the current and future configuration settings our application may have.

Its benefit is that it allows us to manage them as plain and easily readable data on, for example, Hiera YAML files. Such an approach is followed, for example, by the OpenStack modules (<https://github.com/stackforge>), where the configurations of the single components of OpenStack are managed on a settings-based approach, which is fed by the parameters of various classes and subclasses.

For example, the Nova module (<https://github.com/stackforge/puppet-nova>) has many subclasses where there are exposed parameters that map to Nova's configuration entries and are applied via the `nova_config` native type, which is basically a line editing tool that works line by line.

An alternative and quicker approach is to just define a single parameter, such as `config_file_options_hash`, that accepts any settings as a hash:

```
class openssh (
  $config_file_options_hash = { },
)
```

Then manage in a custom template the hash, either via a custom function, like the `hash_lookup()` provided by the `stdmod` shared module (<https://github.com/stdmod/stdmod>):

```
# File Managed by Puppet
[...]
Port <%= scope.function_hash_lookup(['Port', '22']) %>
PermitRootLogin <%= scope.function_hash_lookup(['PermitRootLogin', 'yes']) %>
UsePAM <%= scope.function_hash_lookup(['UsePAM', 'yes']) %>
[...]
```

Or referring directly to a specific key of the `config_file_options_hash` parameter:

```
Port <%= scope.lookupvar('openssh::config_file_options_hash')
['Port'] ||= '22' %>
PermitRootLogin <%= scope.lookupvar('openssh::config_file_options_
hash')['PermitRootLogin'] ||= 'yes' %>
UsePAM <%= scope.lookupvar('openssh::config_file_options_hash')
['UsePAM'] ||= 'yes' %>
[...]
```

Needless to say that Hiera is a good place where to define these parameters; on a YAML based backend we can set these parameters with the following:

```
---
openssh::config_file_template: 'site/openssh/sshd_config.erb'
openssh::config_file_options_hash:
  Port: '22222'
  PermitRootLogin: 'no'
```

Or, if we prefer to use an explicit parameterized class declaration:

```
class { 'openssh':
  config_file_template      => 'site/openssh/sshd_config.erb'
  config_file_options_hash => {
    Port                => '22222',
    PermitRootLogin => 'no',
  }
}
```

Managing multiple configuration files

An application may have different configuration files and our module should provide ways to manage them. In these cases, we may have various options to implement in a reusable module, such as:

- Expose parameters that let us configure the whole **configuration directory**
- Expose parameters that let us configure specific **extra files**
- Provide a **general purpose define** that eases management of configuration files

To manage the whole configuration directory, the following parameters should be enough:

```
class redis (
  $config_dir_path      = $redis::params::config_dir,
  $config_dir_source    = undef,
  $config_dir_purge     = false,
  $config_dir_recurse   = true,
) {
  $config_dir_ensure = $ensure ? {
    'absent' => 'absent',
    'present' => 'directory',
  }
  if $redis::config_dir_source {
    file { 'redis.dir':
```

```

        ensure => $redis::config_dir_ensure,
        path   => $redis::config_dir_path,
        source  => $redis::config_dir_source,
        recurse => $redis::config_dir_recurse,
        purge   => $redis::config_dir_purge,
        force   => $redis::config_dir_purge,
    }
}
}

```

Such a code would allow providing a custom location, on the Puppet Master, to use as source for the whole configuration directory:

```

class { 'redis':
    config_dir_source => 'puppet:///modules/site/redis/conf/',
}

```

Provide a custom source for the whole `config_dir_path` and purge any non managed `config` file: all the destination files not present on the source directory would be deleted: use this option only when we want to have complete control of the contents of a directory:

```

class { 'redis':
    config_dir_source => [
        "puppet:///modules/site/redis/conf--${::fqdn}/",
        "puppet:///modules/site/redis/conf-${::role}/",
        'puppet:///modules/site/redis/conf/' ],
    config_dir_purge => true,
}

```

Consider that the source files, in this example placed in the `site` module according to a naming hierarchy that allows overrides per node or role name, can only be static and not templates.

If we want to provide parameters that allow direct management of alternative **extra files**, we can add parameters like the following (stdmod compliant):

```

class postgresql (
    $hba_file_path           = $postgresql::params::hba_file_path,
    $hba_file_template       = undef,
    $hba_file_content        = undef,
    $hba_file_options_hash   = { } ,
) { [...] }

```

Finally, we can place a **general purpose define** in our module that allows users to provide the content for any file in the configuration directory.

An example can be found at <https://github.com/puppetlabs/puppetlabs-apache/blob/master/manifests/vhost.pp>

Usage is as easy as the following:

```
apache::vhost { 'vhost.example.com':  
  port      => '80',  
  docroot   => '/var/www/vhost',  
}
```

Managing users and dependencies

Sometimes a module has to create a user or have installed some prerequisite packages in order to have its application running correctly.

These are the kind of **extra** resources that can create conflicts among modules, as we may have them already defined somewhere else in the catalog via other modules.

For example, we may want to manage users in our own way and don't want them to be created by an application module, or we may already have classes that manage the module's prerequisite.

There's not a universally defined way to cope with these cases in Puppet, other than the principle of single point of responsibility, which might conflict with the need to have a full working module, when it requires external prerequisites.

My personal approach, which actually I've not seen used around, is to let the users define the name of alternative classes, if any, where such resources can be managed.

On the code side, the implementation is quite easy:

```
class elasticsearch (  
  $user_class      = 'elasticsearch::user',  
) { [...]   
  if $elasticsearch::user_class {  
    require $elasticsearch::user_class  
  }  
}
```

And of course, in `elasticsearch/manifests/user.pp`, we can define the module's default `elasticsearch::user` class.

Modules' users can provide custom classes with:

```
class { 'elasticsearch':  
  user_class => 'site::users::elasticsearch',  
}
```

Or decide to manage users in other ways and unset any class name:

```
class { 'elasticsearch':
  user_class => '',
}
```

Something similar can be done for a dependency class or other classes.

In an outburst of a reusability spree, in some cases I added parameters to let users define alternative classes for the typical module classes:

```
class postgresql (
  $install_class      = 'postgresql::install',
  $config_class       = 'postgresql::config',
  $setup_class        = 'postgresql::setup',
  $service_class      = 'postgresql::service',
  [...] ) { [...] }
```

Maybe this is really too much, but, for example, letting users have the option to define the install class to use, and have it integrated in the module's own relationships logic, may be useful for cases where we want to manage the installation in a custom way.

Managing installation options

Generally, it is recommended to always install applications via packages, eventually to be created onsite when we can't find fitting public repositories.

Still, sometimes we might need, or have to, or want to install an application in other ways, such as just downloading its archive, extracting it and eventually compiling it.

It may not be best practice, but it can still be done, and people do it.

Another reusability feature a module may provide is alternative methods to manage the installation of an application. Implementation may be as easy as the following:

```
class elasticsearch (
  $install_class      = 'elasticsearch::install',
  $install            = 'package',
  $install_base_url   = $elasticsearch::params::install_base_url,
  $install_destination = '/opt',
) {
```

These options expose both the `install` method to use, the name of the installation class (so that it can be overridden), the URL from which to retrieve the archive and the destination to install it in.

In `init.pp`, we can include the `install` class using the parameter that sets its name:

```
include $install_class
```

And in the default `install` class file (here `install.pp`), we can manage the `install` parameter with a case switch:

```
class elasticsearch::install {
  case $elasticsearch::install {
    package: {
      package { $elasticsearch::package:
        ensure => $elasticsearch::managed_package_ensure,
        provider => $elasticsearch::package_provider,
      }
    }
    upstream: {
      puppi::netinstall { 'netinstall_elasticsearch':
        url             => $elasticsearch::base_url,
        destination_dir => $elasticsearch::install_destination,
        owner           => $elasticsearch::user,
        group           => $elasticsearch::user,
      }
    }
    default: { fail('No valid install method defined') }
  }
}
```



The `puppi::netinstall` defined in the above code comes from a module of mine: <https://github.com/example42/puppi> and it's used to download, extract and eventually execute custom commands on any kind of archive.

Users can therefore define which installation method to use with the `install` parameter and they can even provide another class to manage in a custom way the installation of the application.

Managing extra resources

Many times, we have in our environment customizations which cannot be managed just by setting different parameters or names. Sometimes we have to create extra resources, which no public module may provide as they are too custom and specific.

While we can place these extra resources in any class we may include in our nodes, it may be useful to link this extra class directly to our module, providing a parameter that lets us specify the name of an extra custom class, which, if present, is included (and contained) by the module.

```
class elasticsearch (
  $my_class = undef,
) { [...]
  if $elasticsearch::my_class {
    include $elasticsearch::my_class
    Class[$elasticsearch::my_class] ->
    Anchor['elasticsearch::end']
  }
}
```

Another method to let users create extra resources by passing a parameter to a class is based on the `create_resources` function. We have already seen it: it creates all the resources of a given type from a nested hash where they can be defined by their names and arguments. An example from <https://github.com/example42/puppet-network>:

```
class network (
  $interfaces_hash = undef,
  [...] ) { [...]
  if $interfaces_hash {
    create_resources('network::interface', $interfaces_hash)
  }
}
```

In this case, the type used is `network::interface` (provided by the same module) and it can be fed with a hash. On Hiera, with YAML backend, it could look like the following:

```
---
network::interfaces_hash:
  eth0:
    method: manual
    bond_master: 'bond3'
    allow_hotplug: 'bond3 eth0 eth1 eth2 eth3'
  eth1:
    method: manual
    bond_master: 'bond3'
  bond3:
    ipaddress: '10.10.10.3'
    netmask: '255.255.255.0'
```

```
gateway: '10.10.10.1'
dns_nameservers: '8.8.8.8 8.8.4.4'
bond_mode: 'balance-alb'
bond_miimon: '100'
bond_slaves: 'none'
```

As we can imagine, the usage patterns that such a function allows are quite wide and interesting. Being able to base on pure data all the information we need to create a resource may definitively shift on the data backend most of the logic and the implementation that is done with Puppet code and normal resources. But it can also lead to reduced maintainability if used too much; we should use it with defines with a clear responsibility and try to avoid having multiple calls to `create_resources` for the same define.

Summary

In this chapter, we have reviewed Puppet modules, exploring various aspects of their function. We saw how Puppet language evolution has influenced the design of modules, in particular regarding how parameters are exposed and managed, from class parameterization to data in modules. We have also seen common approaches like the `params` and the anchor patterns. We have analyzed the different kinds of parameters a module can expose and where they can be placed. We have also covered the `stdmod` naming convention initiative. We have studied some of the reusability options we can add to modules to manage configuration files, extra resources, custom classes, and installation options.

Now it's time to take a further step and review how we can organize modules at a higher abstraction layer and how people are trying to manage full stacks of applications. This is a relatively unexplored field, where different approaches are still trying to find common consensus and adoption.

6

Higher Abstraction Modules

Most of the modules we can find on the Puppet Forge have one thing in common: they typically manage a single application (Apache, JBoss, ElasticSearch, MySQL and so on) or a system's feature (networking, users, limits, sysctl and so on).

This is a good thing: a rigorous approach to the single responsibility principle is important in order to have modules that can better interoperate, do just what they are expected to do, and behave like libraries that offer well identified and atomic services to their users.

Still our infrastructures are more complex, they require different applications to be configured so as to work together, where configurations may change according to the number and topology of the other components and some kind of cross-application dependencies have to be followed in order to fulfill a complete setup.

This is generally managed by Puppet users when they group and organize classes according to their needs. Most of the time this is done in local site modules which may contain many traps and make the lives of Puppet users more difficult.

The **roles and profiles** pattern described in *Chapter 4, Designing Puppet Architectures*, is a first attempt to formalize an approach to the organization of classes that is based on higher abstraction layers and lets users coordinate the configurations of different modules and make them composable, so that the same elements can adapt to different topologies, application stacks, and infrastructures.

Still, there is much to explore in this field and I think there is much more work to be done.

In this chapter, we will review the following topics:

- Why we need higher abstraction modules
- The OpenStack example
- A very personal approach to reusable stack modules
- Understanding the need for higher abstractions

It took some years in the life of Puppet to achieve the remarkable goal of having good and reusable modules that can be used to quickly and easily install applications and configure them as needed in different contexts.

The Puppet Forge offers quality and variety and, even though many standardization efforts are still needed, both beginners and advanced users can now easily manage most of the applications they need to administer with Puppet.

Still, when it comes to organizing modules and configurations for our infrastructures, documentation, public code and samples are sparse, given the notable exception of the roles and profiles pattern.

The main reason is quite obvious: here is where customizations begin and things get *local*, here is where we use site modules, ENC's and/or a bunch of Hiera data to configure with Puppet our army of servers according to our needs.

The concept of an application stack is obviously not new; we always have to cope with stacks of applications, from simple ones like the well-known Apache, PHP, MySQL (the OS is irrelevant here) plus the PHP application to deploy on top of it, to more complex ones where different components of the stack are interconnected in various ways.

In modern times such stacks are supposed to be composable and elastic, and we can have a (L)AMP stack where the web servers can scale horizontally. We may have a software load balancer that has to account for a dynamic number of web frontends and a backend database which might be supposed to scale too, for example adding new slaves.

We need to manage this on Puppet and here the complexity begins.

It's not difficult to puppetize an existing stack of applications, grouping the classes as expected for our nodes and configuring the relevant software is what we have always done, after all.

This is what is done in role classes, for example, where profiles or module applications are included as needed, typically managing the dependencies in the installation order, if they refer to resources applied to the same node.

It is more difficult to puppetize a dynamic and composable stack, where cross-nodes dependencies are automatically managed and we can change the topology of our architecture easily and add new servers seamlessly. To do this without having to manually review and adapt configuration files, our Puppet classes have to be smart enough to expose parameters to manage the different scenarios.

It's more difficult, but definitively more powerful. Once we achieve the ability to define our components and topology, we can start to work on a higher level, where users don't have to mess with Puppet code but can manage infrastructures with data.

When we have to deal only with data, things may become more interesting, as we can provide data in different ways, for example, via an external GUI, which can restrain and validate users' input and expose only high level settings.

The OpenStack example

The Puppet OpenStack modules (search *Puppet* at <http://github.com/openstack>, formerly hosted in <http://github.com/stackforge>), are probably the largest and most remarkable example of how Puppet is used to manage a complex set of applications that have to be interconnected and configured accordingly.

Component (application) modules

There are different modules for each OpenStack **component** (such as Nova, Glance, Horizon, Cinder, Ceilometer, Keystone, Swift or Quantum/Neutron). They can be retrieved from [, so, for example, Nova's module can be found at <https://github.com/openstack/puppet-nova>.](https://github.com/openstack/puppet-<code><component></code>)

These modules manage all the different configurations via a settings-based approach, with native types that set the single lines of each configuration file (which may be more than one for each component) and with different subclasses that expose all the parameters needed to manage different services or features of each component.

For example, in the Nova module, you have native types like `nova_config` or `nova_paste_api_ini` that are used to manage single lines respectively in the `/etc/nova/nova.conf` and `/etc/nova/api-paste.ini` configuration files.

These types are used in classes like `nova::compute`, `nova::vncproxy` (and many others) to set specific configuration settings according to the parameters provided by users.

There is also a `nova::generic_service` which manages the installation and the service of specific Nova services.

Finally, subclasses like `nova::rabbitmq` or `nova::db::mysql` manage the integration with third-party modules to create, for example, database users and credentials.

A similar structure is replicated on the modules of the other OpenStack components with the added benefit of having a coherent and predictable structure and usage pattern, which makes users' lives much more comfortable.

The general approach followed, therefore, for OpenStack's component modules is to have modules with multiple entry points for data, basically one for each subclass, with configuration files managed with a settings-based approach.

These single component modules can be composed in different ways; we will review a few of them:

- The official and original OpenStack module
- A Puppet Labs module based on roles and profiles
- The full data-driven scenario node terminus-based approach

Raising abstraction – the official OpenStack module

The official and general `openstack` module (<https://github.com/stackforge/puppet-openstack>) can manage the specific roles in an OpenStack infrastructure as it is dedicated to computing, storage, networking or controllers and allow users to quickly and easily define specific (although rather limited) topologies.

For example, the `openstack::all` class manages an all-in-one installation of all the components on a single node, `openstack::controller` installs the OpenStack controller components on a node, and the `openstack::compute` class installs the OpenStack compute components.

All these classes expose parameters that allow users to set public and private addresses, users' credentials, network settings, and whatever is needed to configure, at high level, the whole OpenStack environment. In many cases the parameters and parts of the code are duplicated, for example, the `openstack::all` and `openstack::controller` classes have many parts in common and for each parameter added to the main OpenStack classes there's the need to replicate it on the declared component classes.

This `openstack` module can be definitively considered a module that operates at a higher abstraction layer and uses classes and defines application modules but has some severe limitations on flexibility and maintainability. Its development has been discontinued in favor of Puppet Labs' module that we'll see next.

Raising abstraction – Puppet Labs OpenStack module

Puppet Labs has published another higher abstraction module to manage OpenStack (probably the one used for their internal infrastructure) which is an alternative to Stack Forges `puppet-openstack` but uses the same mainstream component modules.

You can find it at <https://github.com/puppetlabs/puppetlabs-openstack> (there are also versions for earlier OpenStack releases, as `puppetlabs-havana` for Havana or `puppetlabs-grizzly` for grizzly) and it's definitely worth a look as it's also a good real-life example of the application of the roles and profiles pattern.

On your nodes you include the relevant role classes:

```
node /^controller/ {
  include ::openstack::role::controller
}
node /^network/ {
  include ::openstack::role::network
}
node /^compute/ {
  include ::openstack::role::compute
}
```

The role classes include the profile classes and manage a high level dependency order, for example in `openstack::role::controller`, which is the most complex:

```
class openstack::role::controller inherits ::openstack::role {
  class { ' '::openstack::profile::firewall': }
  class { ' '::openstack::profile::rabbitmq': } ->
  class { ' '::openstack::profile::memcache': } ->
  class { ' '::openstack::profile::mysql': } ->
  class { ' '::openstack::profile::mongodb': } ->
  class { ' '::openstack::profile::keystone': } ->
  class { ' '::openstack::profile::swift::proxy': } ->
  class { ' '::openstack::profile::ceilometer::api': } ->
  class { ' '::openstack::profile::glance::auth': } ->
  class { ' '::openstack::profile::cinder::api': } ->
  class { ' '::openstack::profile::nova::api': } ->
}
```

```
class { '::openstack::profile::neutron::server': } ->
class { '::openstack::profile::heat::api': } ->
class { '::openstack::profile::horizon': }
class { '::openstack::profile::auth_file': }
}
```

In the profile classes the required resources are finally declared using both the official OpenStack components modules and classes and defines from other modules. Some of these modules are not part of the OpenStack modules, but are needed for the setup, such as Puppet Labs' firewall module, to manage firewalling, or the MongoDB, MySQL, RabbitMQ, Memcache modules. Here you can also find spot resources declarations like SELinux settings or the usage of `create_resources()` to manage OpenStack's users.

These profile classes don't expose parameters but access configuration data through explicit `hieradata` function calls. Hiera's name spacing is quite clear and well-organized and users can configure the whole setup with YAML files with content like:

```
openstack::region: 'openstack'
##### Network
openstack::network::api: '192.168.11.0/24'
openstack::network::api::device: 'eth1'
openstack::network::external: '192.168.22.0/24'
openstack::network::external::device: 'eth2'
openstack::network::management: '172.16.33.0/24'
openstack::network::management::device: 'eth3'
openstack::network::data: '172.16.44.0/24'
openstack::network::data::device: 'eth4'

##### Fixed IPs (controller)
openstack::controller::address::api: '192.168.11.4'
openstack::controller::address::management: '172.16.33.4'
openstack::storage::address::api: '192.168.11.5'
openstack::storage::address::management: '172.16.33.5'

##### Database
openstack::mysql::root_password: 'spam-gak'
openstack::mysql::allowed_hosts: ['localhost', '127.0.0.1',
'172.16.33.%']

##### RabbitMQ
openstack::rabbitmq::user: 'openstack'
openstack::rabbitmq::password: 'pose-vix'
```

The module supports only RedHat-based distributions and, compared to Stack Forge's official OpenStack module, looks better organized, with less code duplication and some added flexibility.

Taking an alternate approach

Let me tell you a small personal story about Puppet, OpenStack, and choices. I had been asked by a customer to puppetize a multi-region fully HA OpenStack infrastructure.

The customer's internal crew had great OpenStack skills and no Puppet experience, they configured manually an internal testing setup and they wanted to be able to quickly reproduce it on different data centers.

At that times, I had no experience of OpenStack and started to study both it and its existing Puppet modules.

Soon I realized that it would have been quite a pain to reproduce the already existing customer's configuration files with the **settings-based** OpenStack's official modules and that it would have been quite hard for the crew to manage their configurations with a pure data driven approach.

The situation therefore was:

- I didn't know much of OpenStack nor of its Puppet modules
- The customer's team used to work and think on a **files-based** logic
- The OpenStack architecture of the single regions was always the same (given the variable data) and was definitely more complex than the samples in existing modules
- Internal Puppet skills had to be built on an on the job training basis
- Budget and time were limited

I opted for what can be definitely considered a *bad idea*: to write my own OpenStack modules, well aware that I was trying to do it alone in a few days, without even knowing the product, part of which many skilled and knowledgeable people had done in months of collaborative work.

Modules are published on GitHub and are based on the reusability and duplicability patterns I've refined over the years: they all have a standard structure that can be quickly cloned to create a new module with limited effort.

For example, the Nova module (<https://github.com/example42/puppet-nova>) has standard `stdmod` compliant parameters in the `nova` class that allow users to freely manage packages, services, and configuration files. OpenStack components like Nova, may have different services to manage, so I copied the idea of a general purpose `nova::generic_service` definition to manage them and added a useful definition to manage any configuration file: `nova::conf`.

Once defined the basic structure of all the other modules were cloned and adapted to manage the other OpenStack components (which most of the time was just a matter of a few retouches on the module's `params.pp` manifest).

Configurations are delivered via ERB templates, defined as all the classes grouping logic in a single site module. Data is stored on Hiera, and is mostly limited to endpoint addresses, credentials, networking, and other parameters that may change per region or role.

Most of the complexity of an OpenStack configuration is managed as static or dynamic data in templates. The structure is therefore not particularly flexible but reproduces the designed and requested layout.

Adaptation to different topologies is not supposed to be frequent and is a matter of changing templates and logic in the site class.

Tuning of configurations, which might be more frequent, is a matter of changing the ERB templates, so it is probably easier for the local crew to manage them directly.

Frankly, I'd hardly recommend the usage of these modules to setup an OpenStack infrastructure, but they can be useful to replicate an existing one or to manage it in a file-driven way.

The general lesson we can take from this is the usual golden one, especially true in the Puppet world: there's never a best way to do things, but there is the most appropriate way for a specific project.

An approach to reusable stack modules

What we have seen up to now are more or less standard and mainstream Puppet documentation and usage patterns. I have surely forgotten valuable alternatives and I may have been subjective on some solutions, but they are all common and existing ones, nothing has been invented.

In this section, I'm going to discuss something that is not mainstream, has not been validated in the field, and is definitely a personal idea on a possible approach to higher abstraction modules.

It's not completely new or revolutionary, I'd rather call it evolutionary, in the line of established patterns like parameterized classes, growing usage of PuppetDB, roles and profiles, with a particular focus on reusability.

I call **stack** here a module that has classes with parameters, files, and templates that allow the configuration of a complete application stack, either on a single all-in-one node or on separated nodes.

It is supposed to be used by **all** the nodes that concur to define our application stack, each one activating the single components we want to be installed locally.

The components are managed by normal application modules, whose classes and definitions are declared inside the stack module according to a some what opinionated logic that reflects the stack's target.

In my opinion there's an important difference between application (or component) modules and stack (higher abstraction) modules.

Application modules are supposed to be like reusable libraries; they shouldn't force a specific configuration unless strictly necessary for the module to work. They should not be opinionated and should expose alternative reusability options (for example, different ways to manage configuration files, without forcing only a settings or file-based approach).

Stack modules have to provide a working setup, they need templates and resources to make all the stuff work together. They are inherently opinionated since they provide a specific solution, but they can present customization options that allow reusability in similar setups.

The stack's classes expose parameters that allow:

- High-level setting of the stack's main parameters
- Triggers to enable, or not, single components of the stack
- The possibility to provide custom templates alternative to the default ones
- Credentials, settings, and parameters for the relevant components

Let's look at a sample `stack::logstash` class that manages a logging infrastructure based on LogStash (a log collector and parsing tool), Elasticsearch (a search engine), and Kibana (a web frontend for Elasticsearch). This is obviously an opinionated setup, even if it is quite common for LogStash.

The class can have parameters like:

```
class stack::logstash (
    $syslog_install = false,
```

```
$syslog_config_template      = 'stack/logstash/syslog.conf.erb',
$syslog_config_hash          = { },
$syslog_server               = false,
$syslog_files                = '*.*',
$syslog_server_port          = '5544',

$elasticsearch_install       = false,
$elasticsearch_config_template = 'stack/logstash/elasticsearch.
yaml.erb',
$elasticsearch_config_hash    = { },
$elasticsearch_protocol      = 'http',
$elasticsearch_server        = '',
$elasticsearch_server_port    = '9200',
$elasticsearch_cluster_name   = 'logs',
$elasticsearch_java_heap_size = '1024',
$elasticsearch_version        = '1.0.1',

$logstash_install            = false,
$logstash_config_template    = 'stack/logstash/logstash.conf.erb',
$logstash_config_hash        = { },

$kibana_install              = false,
$kibana_config_template      = undef,
$kibana_config_hash          = { },
) {
```

You can see some of the reusability oriented parameters we have discussed in *Chapter 5, Using and Writing Reusable Modules*, the class' users can provide:

- **High level parameters** that define hostnames or IP addresses of the infrastructure components (if not explicitly provided, the module tries to calculate them automatically via PuppetDB) as `syslog_server` or `elasticsearch_server`
- **Custom ERB templates** for each managed application that overrides the default ones as `syslog_config_template` or `logstash_config_template`
- **Custom hash of configuration settings**, if they want a fully data-driven setup (they need to provide templates that use these hashes) as `logstash_config_has`

For each of the managed components, there's a Boolean parameter that defines if such a component has to be installed (`elasticsearch_install`, `logstash_install` ...).

The implementation is quite straightforward, if these variables are `true`, the relevant classes are declared with parameters computed in the stack class:

```

if $elasticsearch_install {
  class { 'elasticsearch':
    version      => $elasticsearch_version,
    java_opts    => $elasticsearch_java_opts,
    template     => $elasticsearch_config_template,
  }
}

if $syslog_server and $syslog_install {
  if $syslog_config_template {
    rsyslog::config { 'logstash_stack':
      content => template($syslog_config_template),
    }
  }
  class { '::rsyslog':
    syslog_server => $syslog_server,
  }
}

```

The resources used for each component can be different and have different parameters, defined according to the stack class' logic and the modules used.

It's up to the stack's author as to the choice of which vendors to use for the application modules and how many features, reusability options, and how much flexibility to expose to the stack's users as class parameters.

The stack class(es) are supposed to be the only entry point for users' parameters and they are the places where resources, classes, and definitions are declared.

The stack's variables, which are then used to configure the application modules, can be set via parameters or calculated and derived according to the required logic.

This is a relevant point to underline: the stack works at a higher abstraction layer, and can manipulate and manage how interconnected resources are configured.

At the stack level you can define, for example, how many ElasticSearch servers are available, what are the LogStash indexers and how to configure them in a coherent way.

You can also query PuppetDB in order to set variables based on your dynamic infrastructure data.

In this example the `query_nodes` function from the `puppetdbquery` module (we have seen it in *Chapter 3, Introducing PuppetDB*) is used to fetch hostnames and IP addresses of the nodes where the `stack` class has installed ElasticSearch. The value retrieved from PuppetDB is used if there isn't an explicit `$elasticsearch_server` parameter set by users:

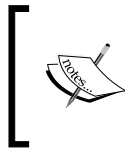
```
$real_elasticsearch_server = $elasticsearch_server ? {  
  ''      => query_nodes('Class[elasticsearch]', ipaddress),  
  default => $elasticsearch_server,  
}
```

In this case the `stack` manages configurations via a file-based approach, so it uses templates to configure applications.

The `stack` class has to provide default templates, which should be possible to override, where the `stack`'s variables are used. For example, `stack/logstash/syslog.conf.erb` can be something like:

```
<%= scope.lookupvar('stack::logstash::syslog_files') %> @@<%= scope.lo  
okupvar('stack::logstash::syslog_server') %>:<%= scope.lookupvar('stac  
k::logstash::syslog_server_port') %>
```

Here the `scope.lookupvar()` function is used to get variables by their fully qualified name so that they can be consistently used in our classes and templates.



Note that such a model requires all the used application modules to expose parameters that are allowed for its users (in this case the `stack`'s developer), providing the possibility to use custom templates.

When using Hiera, the general `stack`'s parameters can be set in `common.yaml`:

```
---  
stack::logstash::syslog_server: '10.42.42.15'  
stack::logstash::elasticsearch_server: '10.42.42.151'  
stack::logstash::syslog_install: true
```

Specific settings or install Booleans can be specified in role-related files such as `el.yaml`:

```
---  
stack::logstash::elasticsearch_install: true  
stack::logstash::elasticsearch_java_opts: '-Xmx1g -Xms512m'
```

Compared to profiles, as commonly described, such stacks have the following differences:

- They expose parameters, so a user's data directly refers to the stack's classes
- The `stack` class is included by all the nodes that concur with the stack, with different components enabled via parameters
- Cross-dependencies, order of executions, and shared variables used for different applications are better managed at the stack level, thanks to being a unique class that declares all the others
- The stack allows the decoupling of user data and code from the application modules, which enables changes to the application module implementations without touching the user-facing code

 A possible limitation of such an approach is when the same node includes different stack classes and has overlapping components (for example, an `apache` class). In this case the user should manage the exception disabling the declaration of the `apache` class, via parameters, for one of the classes.

Tiny Puppet

As we can see when developing Puppet modules, when we want to configure a service, whatever it is, we follow more or less the same pattern: we install the software, we deploy the configuration files, and we start the services.

Tiny Puppet (<http://tiny-puppet.com/>) is an ambitious project that aims to provide a unified way to manage any kind of software in any operating system using minimalistic code by taking advantage of this common pattern that most modules implement.

The project is basically divided into two parts. One is a smart module with a collection of definitions that can manage the basic use cases: packages installation, configuration files management, and additional repositories setup. The second part is `tiny data`, a repository of Hiera data that defines the characteristics of each supported software so the first module knows how to manage it.

Then, for example, to install an Apache server in any of the supported operating systems, we only need the `tiny Puppet` module and a line of Puppet code:

```
tp::install { 'apache': }
```

If we need to provide additional configurations, we can use the `tp::conf` define:

```
tp::conf { 'apache::example.com.conf':  
  template => 'site/apache/example.com.conf.erb',  
  base_dir => 'config', # Use the settings key: config_dir_path  
}
```

As you can see, we are not saying which package installs Apache, or if it contains a service, and we are also not telling `tp::conf` where to place the configuration file. Here is where the second part of the project, tiny data, enters into the game. For example, for the Apache project it contains some default settings and additional overrides for specific operating systems. For example, the YAML file with the Hieradata with the default settings looks like the following:

```
---  
apache::settings:  
  package_name: 'httpd'  
  service_name: 'httpd'  
  config_file_path: '/etc/httpd/conf/httpd.conf'  
  config_dir_path: '/etc/httpd'  
  tcp_port: '80'  
  pid_file_path: '/var/run/httpd.pid'  
  log_file_path: [ '/var/log/httpd/access.log' , '/var/log/httpd/  
error.log' ]  
  log_dir_path: '/var/log/httpd'  
  data_dir_path: '/var/www/html'  
  process_name: 'httpd'  
  process_user: 'apache'  
  process_group: 'apache'
```

And the overrides for FreeBSD are:

```
---  
apache::settings:  
  config_file_path: '/usr/local/etc/apache20/httpd.conf'  
  config_dir_path: '/usr/local/etc/apache20'  
  config_file_group: 'wheel'
```

Some of the more useful implemented defines are:

- `tp::install`: Installs the application, and manages its service, if any
- `tp::conf`: Manages specific configuration files, multiple directories, where to place them and can exist for any software
- `tp::dir`: Provides full directories with configuration, which can also be taken from remote repositories
- `tp::repo`: Configures additional repositories for supported applications

Summary

Puppet modules are getting better and better, they are more reusable, they better fit doing just what they are supposed to do, and they offer stable and reliable interfaces to the management of different applications.

They are therefore getting nearer to that status where they can be considered shared libraries that can be used by different users to compose the configurations needed in their environments.

Here is where many people are struggling in order to achieve a sane organization of resources, good patterns to group them, and better approaches to a dynamic, reproducible, structured, and maybe reusable management of complete stacks of applications.

At this higher abstraction layer module, people are experimenting with different practices. Some have become common, like the roles and profiles pattern, but still few are engineered with the vision to be reusable and eventually allow single components to be composed freely to fit different topologies.

In this chapter, we focused on this topic, reviewing why we need higher abstraction modules and why it would be great to have them reusable and flexible.

There's still much to do on this topic and I think much will be done in future years.

In the next chapter, we move to completely new topics. We will explore the practical challenges we face when we have to introduce Puppet, either on a new or an existing infrastructure.

7

Puppet Migration Patterns

We're probably already working with Puppet or we are planning to use it in the near future. In any case, we have to face the daunting task of automating the configuration and reproducing the state of a variable number of different servers.

Using Puppet for server infrastructure involves analysis, planning, and decisions.

In this chapter, we will review the different steps to be considered in our journey towards infrastructure automation:

- The possible scenarios and approaches we might face: new setups, migrations and updates of existing systems, and Puppet code refactoring
- Patterns for Puppet deployments; a step-by-step approach divided in reiterated phases: information gathering, priorities definition, evaluation on how to proceed, coding, testing, applying the changes, and verifying the reports
- The cultural and procedural changes that configuration management involves

Examining potential scenarios and approaches

Puppet is a tool that offers a world of order, control, and automation; we have learnt it, we know at least the basics of how it works and we want to install it on our servers.

We might have just a few dozens or several thousand systems to manage, they might be mostly similar or very different from one another, have very few things to be configured or a large number of resources to deal with.

We might manage them by hand or, maybe, already have some automation scripts or tools, Puppet itself included, that help us in installing and maintaining them but do that in a suboptimal way that we want to fix.

We might have a brand new project to build and this time we want to do the right thing and automate it from the start.

The number of variables here are many, every snowflake is unique, but there's a first basic and crucial distinction to make; it defines where our work will be done:

- On a brand **new servers infrastructure**
- On **existing systems**, either managed manually or already (semi-automated)
- On an infrastructure **already managed by Puppet**, to be heavily refactored

This distinction is quite important and can affect heavily how quickly and safely we can work with Puppet.

New infrastructures

If we work on a startup or a new project, we have the opportunity to create our infrastructure from scratch, which is often a preferred, easier, more comfortable, and safer option for various reasons:

- **More freedom on architectural choices:** Starting from scratch we are not bound to existing solutions, we can concentrate on the tools and languages that better fit our current and future needs without caring about backwards compatibility and existing legacies. This freedom does not strictly relate to just our Puppet activities and it allows us to evaluate technologies also based on how easily they can be automated.
- **Brand new operating system and application stacks, possibly homogeneous:** We can choose a recent version of our preferred OS and, even better, we can start from a standardized setup where all servers have the same OS and version which definitely makes our life with Puppet easier and our infrastructure more coherent and easier to manage.
- **Clean setups:** We don't have to be concerned about, reproduce, reverse engineer, and maintain existing systems, so we know exactly the configurations needed on them. There is always the risk of messing with them in the future, but introducing a configuration management system at the beginning definitely helps in keeping things tidy and homogeneous for a long time.

- **Sound design from the foundations can be defined:** We can setup our servers using all our expertise on how to do things in a good way and we can generally benefit from the fact that newer versions of applications generally make their setup easier and easier, for example with updated packages, options that make configuration easier or allow splitting files in `conf.d` directories.
- **No mess with current production:** Being a new project, we don't have to cope with existing running systems and we don't risk introducing disrupting changes while applying or testing our Puppet configurations.
- **Faster and smoother deployment times:** These are the natural consequences of being free from working on existing setups in production. We don't have to worry about maintenance windows, safe runs, double checks, remediation times, and all the burdens that can greatly reduce the pace of our work.

Existing manually managed infrastructures

When we need to automate existing systems things are quite different, as we have to cope with stratifications of years of many people's work with different skills under variable conditions. They may be the result of a more or less frantic and inorganic growth and evolution of an IT infrastructure, whose design patterns may have been changed or evolved with time. This might not be the case for everybody, but is quite common with infrastructures where a configuration management tool has not been steadily introduced.

We will probably have to manage:

- A **heterogeneous** mess of different **operating systems** and versions, deployed over many years. It is indeed very difficult to have OS homogeneity on a mature infrastructure. If we are lucky, we have different versions of the same OS. This is because they were installed at different times, but more likely we have to manage a forest of variegated systems, from Windows to Unix to almost any conceivable Linux distribution, installed according to the project software needs, commercial choices, or personal preferences of the sys admins in charge at the time.
- **Heterogeneous hardware**, from multiple vendors and of different ages, is likely to be present in our infrastructure. We have probably already made great progress in recent years, introducing virtualization and maybe a large effort of consolidation of the previous hardware mess has already been done. Still, if we haven't moved everything on the cloud, it's very likely that our hardware is based on different models possibly by different vendors.

- **Incoherent setups**, with systems configured by different people who may have used different logic, settings, and approaches also for the same kind of functionality. Sys admins might be complex and arcane dudes, who like to make great things in very unexpected ways. Order and coherency does not always find a place on their crazy keyboards.
- **Uncertain setup procedures** whose documentation may be incomplete, obsolete, or just non-existent. Sometimes we have to reverse engineer them, sometimes the offered functionality is easy to understand and replicate, other times we may have an arcane black box, configured at the dawn of time by people who don't work anymore in the company, which nobody wants to touch because it's too critical, fragile or simply indecipherable.
- **Production systems**, which are always delicate and difficult to operate. We might be able to work only on limited maintenance windows, with extra care of the changes introduced by Puppet and their evaluation on operations. We have to make choices and decisions for each change, as the homogeneity that Puppet introduces will probably change and leverage some configurations.
- **Longer Puppet rollout** times are the obvious side effects when we work on existing systems. These have to be taken into account, as they might influence our strategy for introducing Puppet.

We have two possible approaches to follow when dealing with existing systems. They are not mutually exclusive, we can adopt both of them in parallel, evaluating for each node which one is the most fitting:

- **Node migration:** We make new servers managed by Puppet that replace the existing ones
- **Node update:** We install Puppet on existing machines and start to manage them with Puppet

Node migration

A migration approach involves the installation and configuration, via Puppet, of new servers as replacements for existing ones: we build a system from scratch, we configure it so that it reproduces the current functionalities and, when ready, we promote it to production and decommission the old machine.

The main advantages of such an approach, compared to a local update, are as follows:

- We can **operate on new systems** that are not (yet) in production, so we can work faster and with less risk. There is no need to worry about breaking existing configurations or introducing downtimes to production.

- The **setup procedure is done from scratch**, so we're validating whether it's fully working and whether we have full Puppet coverage of the required configurations. Once a system has been successfully migrated and fully automated, we can be confident that we can consistently repeat its setup any time.
- We can **test our systems** before decommissioning the old ones and a rollback is generally possible if something goes wrong when we make the switch.
- We can use such an occasion to **rejuvenate our servers**, install newer and patched operating systems and generally have a more recent application stack (if our migrated applications can run on it), although it's always dangerous to do multiple changes at the same time.

There is a **critical moment** to consider when we follow this approach: when we **shift** from the old systems to the new ones.

Generally this can be done with a change of IP on the involved systems (with relevant ARP cache cleanups on peer devices on the same network), or of DNS entries or in configurations of load balancers or firewalls; all these approaches are reversible being mostly done at network level and rollback procedures can be automated for a quick recovery in case of problems.

Whatever the approach used we'll have the new systems doing what was previously done by old and different ones, so we might have untested and unexpected side effects. Maybe without the commonly used features of an application, or scheduled jobs which might manifest themselves at later and unexpected times.

So it makes sense to keep the migrated systems under observation for some days and double check if, besides the correct behavior of the mains services they provide, other functionalities and interactions with external systems are preserved.

Generally the shift is easier if no data is involved; much more attention is definitely needed when migrating systems that persist data, such as databases.

Here the scenarios might be quite different and the strategy to follow for data migration depends on the software used, the kind and size of data, if it's stored locally or via a network, and other factors that definitely are out of the scope of this book.

Node update

When we prefer or have to install Puppet on existing running systems, the scenario changes drastically. Different variables come into play and the approaches to follow are definitely alternative.

In these cases, we will have:

- Probably **different OS** to manage, some of them might be rather ancient and here we might have problems in installing a recent version of Puppet, so it makes sense to verify if the version of the client is compatible with the one of the Puppet Master.
- **Undetermined setup and configuration procedures** that we might have to gradually cover with Puppet with the hope of managing whatever is needed.
- Puppet deployments on **running production systems** so we have to manage the risk of configuration changes delivered by Puppet. This makes our lives harder and more dangerous, especially if we don't follow some precautions (more on this later).
- **Manual configurations** accumulated over time will probably determine systems which are not configured in the same way and we'll realize this when Puppet is deployed. What's likely to happen is that many unexpected configurations will be discovered and, probably, wrong or suboptimal settings will be fixed.

Existing automated infrastructures

What has been described in the previous paragraphs might have been a common situation some years ago, now times have definitely changed and people have been using configuration management tools and cloud computing for years.

Actually, it is likely that we might already have a Puppet setup and we are looking for ways to reorganize and refactor it.

This is becoming a quite common situation: we learn Puppet while we work on it and during the years of usage patterns evolve, the product offers new and better ways to do things, and we understand better what has worked and what hasn't.

So we might end up in scenarios where our Puppet architecture and code is in different shapes:

- We have some old code that needs **fixing and refactoring** in a somewhat limited and non-invasive way: no revolutions, just evolutions
- Our setup works well but is based on **older Puppet versions** and the migration needs some more or less wide refactoring (a typical case here is to do major version upgrades with backwards incompatibilities)

- Our setup more or less does its job but we want to introduce **new technologies** (for example Hiera) or reorganize radically our code (for example, introducing well separated and reusable application modules and/or the roles and profiles pattern)
- Our Puppet code is a **complete mess**, it produces more problems than the ones it solves and we definitely need a complete rewrite

Different cases may need different solutions, they can follow some of the patterns we describe for the setup of new infrastructures, or the migration or update of existing ones, but here the work is mostly done on the Puppet Master rather than the clients, so we might follow different approaches:

- Refactoring of the code and the tools keeping the **same Puppet Master**. If there are few things to be changed, this can be based on simple evolutionary fixes on the existing code base, if changes are more radical we can use Puppet's **environments** to manage the old and new codebase in parallel and have clients switching to the new environment in a gradual and controlled way.
- Creation of a **new Puppet Master** can be an alternative when changes are more radical, either for the versions and tools used or for the complete overhaul of the codebase. In these cases we might prefer to keep the same certificate authority and Puppet Master's certificate or create a new one, with the extra effort of having to re-sign our clients' certificates. Such an approach actually may involve both the creation of new systems that point directly to the new Puppet Master, or the reconfiguration of existing ones. In both cases we can follow the same gradual approach described for other scenarios. When we need to upgrade the Puppet Master a few rules have to be considered:
 - The Puppet Master version should always be newer than the version on the clients. The client-server compatibility between single major releases is generally guaranteed until Puppet 4. 2.7 servers are backwards compatible with 2.6 clients and 3.x servers are compatible with 2.7 clients. In most cases backwards compatibility extends for more versions, but that's not guaranteed. Starting on Puppet 4 compatibility is only guaranteed for packages in the same package collection, it will make it easier to be sure that even different versions will work well together if they belong to the same collection, but it breaks compatibility with older versions.

- It's recommended to upgrade from one major version to the next one, avoiding wider jumps. Starting from version 3 a semantic versioning scheme has been introduced (earlier major versions can be considered 2.7, 2.6, 0.25, 0.24) and a major version change, now expressed by the first digit, can have backward incompatibilities regarding the language syntax and other components that are not related the client-server communication. Puppet Labs, however, has the good habit of logging deprecation notices for features that are going to be removed or managed in different ways, so we can have a look on the Puppet Master's logs for any similar notice and fix our code before making the next upgrade step.

Patterns for extending Puppet coverage

Let's further analyze what to do when we face the daunting task of introducing Puppet on an existing infrastructure.

Some of these activities are needed also when working on a brand new project or when we want to refactor our existing Puppet configurations; consider them as a reference you can adapt to your own needs and scenario.

The obvious assumption is that Puppet deployment on something that is already working and serving services has to be **gradual**, we have to understand the current situation, the expected goals, what is quick or easy to do, where and what makes sense to manage with Puppet and our operational priorities.

Raising the bar, step by step

A gradual approach involves a **step-by-step** process that needs informed decisions, reiterations and verification of what is done, evaluating each case according to its uniqueness.

The Puppet coverage of our infrastructure can be seen in two dimensions, on one side the number of nodes managed by Puppet, and on the other side the percentage of configurations managed by Puppet in these nodes. So we can extend our Puppet coverage following two axes:

- **Vertically:** Working on all the existing Puppet managed nodes and adding, step by step, more and more resources to what is managed by Puppet, so that at the end of the process we can have full coverage both on the common baselines and the role's specific features. Here, it is better to make a single change at a time, and propagate it to the whole infrastructure.

- **Horizontally:** Adding new nodes under Puppet management, either with a migration or with an update approach. Here, operations are done node by node, they are placed under Puppet control and the baseline of common resource (plus eventually role specific ones) is applied in a single run.

The delivery process should be iterative and can be divided in to five phases:

- **Collect info:** We need to know the territory where we operate
- **Set priorities:** Define what is more important and urgent
- **Make choices:** Evaluate and decide how to proceed
- **Code:** Write the needed Puppet manifests and modules
- **Apply changes:** Run Puppet and check for reports

The first iterations will probably concentrate on developing the **common baseline** of resources to apply to all the nodes. Then these configurations can be propagated both horizontally and vertically, looping on the above phases with more or less emphasis on some of them according to the specific step we are making. Each iteration might last days or a few minutes.

Let's explore more deeply what happens during the phases of each reiteration.

Knowing the territory

Before starting to run Puppet on an existing server's infrastructure, we need to know it.

It sounds obvious, it is obvious.

So obvious that sometimes we forget to do it properly.

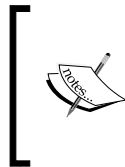
We have to know what we want to automate and we must gather as much info as possible about:

- **How many** servers are involved
- What are their **operating systems** and their major versions
- What are the **common configuration files** and resources to manage on a basic system (our general baseline)
- How many different **kinds of server** we have (our Puppet roles)
- What are the main **applications stacks** used
- What are the most **critical systems**
- What servers are going to be **decommissioned** soon
- What kind of servers we will need to **deploy in the future**

We may be able to find much of this information from an existing inventory system or internal documentation, so it should not be hard to gather it and have a general idea of the numbers involved.

For each application or system resource that we plan to manage with Puppet we also need to know further details like:

- **How configurations change** on different servers according to infrastructural factors like the role of the server, operational environment, datacenter, or zone. We will use variables based on these factors to identify nodes and differentiate configurations according to them.
- How differently the same configuration is **managed on our operating systems**. We will need to manage these OS differences in our modules and Puppet code.
- If there are **unique or special cases**, where the configuration is done differently, we will manage these exceptions on a per node basis.



It's definitely not necessary to gather all this info since at the beginning, some details are useful just when we need to automate the relevant components. What's important at the beginning is the general overview about the systems involved, their OS, roles and life expectancy.

Defining priorities

We said that automation of an existing infrastructure is a gradual process, where we start from 0% of systems and configuration coverage and step by step we move them to a state where they are *Puppet managed*. We might not reach a full 100% coverage, and for each step we might consider what's worth managing with Puppet and what's not.

Ideally, every server of our infrastructure should be fully managed by Puppet and we should be able to provision a system from scratch and have it up and running after a Puppet run: the setup procedure is fully automated, controlled, replicable, and scalable.

There are cases anyway where this rule is not necessarily recommended. When the setup procedure of an application requires execution of few specific commands, with varying arguments that might not be easily predictable or which needs users' interactivity, automation becomes difficult and expensive, in terms of implementation times.

If such setups have to be done only once on a system (during installation time), and are supposed to be done on very few systems that have a medium to long life, we may question if it makes sense to try to do it with Puppet, spending several hours on what can be quickly done manually in few minutes.

This becomes even more evident when we have to manage with Puppet an existing system where such a setup is already done, or when the version of our application may change in the future and the setup procedure will be different, frustrating all our efforts to fully manage it with Puppet.

So, when we face the big task of automating an infrastructure, we definitely have to set priorities and make choices, decide what makes sense to automate first and what can be postponed or simply ignored.

We have to consider different tasks such as:

- **Automate server's deployment:** This is one of the most important things to do, and there is a big chance that we're already managing it in some way. The installation of an operating system should be as quick and automated as possible. This actually is not strictly related to configuration management as it's a step that occurs before we install and run our configuration agent. There are countless ways to automate installations on bare metal, virtual machines, or cloud instances. Whichever is the one we choose, the general rule is that OS provisioning should be quick and essential: we can manage all the specific settings and packages of a given server type via our configuration management tool, which is invoked in a second moment and is independent of the method we've used to install our system.



There's at least an exception for this general rule: if we need to provision very quickly new systems, for example in an AWS auto scaling infrastructure, we might prefer to use AMIs that have already been installed and more or less partially configured the needed applications. This makes the spin up of a cloud instance definitely faster.

- **Automate common systems configurations:** Here we enter into Puppet realm. We may have very different servers fulfilling different tasks but it is very likely that we'll need for each of them a common baseline of shared configurations. The specific settings may change according to different conditions but the configurations to manage are always the same. They typically involve configuration of the DNS resolver and network configuration, user authentication, NTP and time zone settings, monitoring agents, log management, routine system tasks, backup setup, configuration of OpenSSH, the Puppet client setup itself, and whatever other configuration we may want to manage on all our systems. To cover these settings should be our first priority, once completed the benefits are real as they allows us to quickly rollout changes that, if done manually, would probably involve a lot of wasted time.
- **Automate the most important roles:** This is the next big step in our journey to automation; once we've managed the common ground, it is time to raise the bar and start to cover the roles we have in our infrastructure. Here things may become more difficult since we start to manipulate directly the software that delivers the services of our systems. It makes sense to start from the **most used** ones evaluating also **how quickly and easily** they can be managed and how often we are going to setup new ones in the near future. An important element to consider is indeed what are the kind of servers we need to setup in the **future**. It's probably better to work first on the automation of roles that we need to deploy from scratch before taking care of roles that are already in production.
- **Refine application deployments then automate them:** We don't need and actually should not manage application deployments via Puppet, but we can configure with Puppet the environment of our deployment tool, be that MCollective, Capistrano, Run Deck, Puppi, the native OS package system or any other. We might manage deployments with custom scripts or maybe manually, executing a sequence of more or less documented commands, and in such a case that's definitely a thing to fix. Puppet can help but is only a part of the solution. We should concentrate on how to manage the delivery of our deployment via the simple execution of a single command, even executed manually from the shell. Once we achieve this, we can manage that execution with whatever orchestration, remote execution, or continuous integration tool we have in house or we want to introduce, but this can be considered a subsequent evolutionary step. The very first prerequisite is that we should be able to deliver the deployment of our applications with a single command (it may be a script that runs a sequence of other commands, of course) that doesn't involve human interaction.

Puppet here can help to setup and configure things but, I stress it again, it should not be used to actually trigger the execution of such a command. That's not its task or its function: Puppet sets and maintain the systems' state, it is not made to trigger one-shot commands.

- **Integrate what already works well:** This is an important point. Our infrastructure is the result of years of work and effort, the sweat and blood of many sys admins. It might be a total chaos or may stay in good shape, but probably there's something (or much) good there. We should consider Puppet as a tool that automates with a formal language what we are already doing, either with other kinds of automation procedures or manually. We have to review what is working well in our setup procedures and what is not working, what doesn't cause any problems and what drains a lot of time or fails often. We can start to work on what wastes our time and works in a suboptimal way, and keep and integrate in Puppet what does its job well. For example, if we have custom scripts that install and configure a complex application stack, we might consider directly using them in our manifests (with a plain `exec` resource) instead of recreating under Puppet all the configurations and resources that the scripts deliver and build. We can always fix this later, when other priorities have been tackled and there's time for wider refactoring.
- **Automate monitoring:** Monitoring is a beast of its own in a complex infrastructure. There are different things to monitor (servers, services, applications, network equipment, hardware and software components), different kinds of tasks (checking, alerting, trends graphing), different approaches (on premise, SaaS, via local agents or remote sensors) and definitely different tools that do the job. We probably use different solutions according to what and how we need to monitor and we are probably managing them in different ways. Maintenance of the checks and the metrics to be monitored can be a time wasting effort, in some cases, and here is where it makes sense to use Puppet to automate long and painful manual operations. Management and automation of a monitoring infrastructure can be rather complex if the used tool doesn't provide mechanisms of self-discovery. Nagios, for example, probably the most loved, hated, and used monitoring tool, might be a real pain to maintain manually but at the same time is not too easy to manage with Puppet, since it requires external resources and it expects us to define with Puppet code all the desired checks. Given its complexity, automation of monitoring activities might not be at the top of our to do list, but since it's often a time consuming task, we have to face it, sooner or later, especially if we are planning the installation of many new servers, for which we want to automate not only their setup but also their monitoring.

Evaluating solutions

When we know our territory and have set priorities on what to do, we have more opportunities to make decisions. There are many choices to ponder but we don't need to make them all at once. Let's start from the priorities we've set and define how to behave case by case.

Among the decisions to make, we have to evaluate:

- **If Puppet can or should be installed on old or arcane systems:** This is not such a problem with Puppet 4, as packages include their own ruby interpreter. But it can be a problem with older versions or when we cannot install Puppet from packages. Here it's important to understand what is the newest version of Puppet that can be installed (compatibility with the native Ruby version is important here (check <http://docs.puppetlabs.com/guides/platforms.html#ruby-versions> for details) and how this can affect the version of Puppet on the Puppet Master, according to what we have seen about client-server compatibility.
- **Where it makes sense to install Puppet:** Old servers which require low maintenance, provide services that are planned to be decommissioned soon and do not have not a long life expectancy might not deserve our attention and might be skipped or managed with low priority.
- **The strategy to follow for each node - migration or update:** There are no strict rules on this. Both the alternatives are possible and can be evaluated for each host, case by case. Old but needed systems, on obsolete hardware, based on OS not widely spread in our infrastructure, whose configurations might be easy to automate, are good candidates for migration. The alternative update path might be preferred on our most common existing operating systems so that the time spent on implementing configurations with Puppet can be reused multiple times.

- **How easily and quickly configurations can be done:** Activities that we know to be rather quick to implement and present low risks in their deployment can be delivered with precedence, both to reduce our backlog of things to do and to give us the feeling that something is moving. While deploying them we can also gather confidence on the procedures and know our systems better.
- **Impact and risks have always to be accounted:** Puppet is going to *change* configurations, in some cases it might just add or remove some new lines or comments, in other cases it will actually change the content of configurations. Doing this it will probably restart the relevant services and have in any case an impact on our systems. Is it safe? Can be done at any time? What happens if something goes wrong? What kind of side effects can we expect from a change? All of these questions need a more or less comprehensive answer, and this has to be done for each step. Incidentally, this is a reason why each step should be limited and introduce only a single change: it will be easier to control its application and diagnose eventual problems.
- **Current maintenance efforts and systems stability:** This should be considered when defining how to proceed and what to do first. If we are losing a lot of time doing repetitive tasks that can be automated via Puppet, or we keep on firefighting recurring failures on some troublesome systems, we should definitely concentrate on them. In this case the rule of doing first what is easier to do has to be coupled with another common sense rule: automate and fix first what causes a lot of wasted time.

Coding

We have gathered info about our systems, set priorities, and evaluated solutions. Now it's finally time to write some code. The previous steps may have involved a variable amount of time, from a few minutes to days. If we have spent too much time on them, we have probably done something wrong, or too early, or we might have made too much upfront planning instead of a step-by-step approach where the single phases reiterate relatively quickly.

In any case, when we need to write Puppet code in line with our priorities we should:

Develop the common baseline for all our systems: This is probably the first and most useful thing to do. We can start to cover the most important and used OS in our infrastructure, we don't need to manage at the beginning the general baseline of all our servers. The common classes and resources can be placed in a grouping class (something like `::site::general`) which will likely include other classes which might be both from shared modules (`::openssh`, `::ntp`) or from custom ones (`::site::users`, `::site::hardening`). It makes sense also to include in `::site::general`, a class that includes a subset of common resources, something we can call `::site::minimal`. This can be useful to let us install Puppet with minimal configurations (eventually just what's needed to manage the same Puppet client and nothing more) on as many servers as possible and then gradually extend coverage of new systems configurations just by moving resources and classes from `::site::general` to `::site::minimal`. These are just samples on how practically a common baseline might be introduced and mileages may vary.

- **Develop the most used roles:** Once the common baseline is configured we can concentrate on developing full roles, starting from the most useful, easy to manage and common ones. This might take a varying amount of time, but the general idea is that each new node should be fully managed. It doesn't make sense to introduce technical debts also on the brand new nodes we have to deploy that are managed by Puppet. So, if we had to choose between automating the existing or new servers, we should give precedence to the latter ones.
- **Each new node should be fully managed:** It doesn't make sense to introduce technical debts on the brand new nodes we have to deploy that are born managed by Puppet. So if we had to choose from automating existing or new servers, we should give precedence to the latter ones.

Besides the macro activities that involve high level planning, we will face at every step some micro decisions on how to manage the configurations for a given application on our servers. There are some common rules to consider and they replicate the phases we have described that involve information gathering and decisions on the approaches to follow.

We must know how the configuration file(s) of an application changes on our systems and this is generally affected by two factors: **operating system** and **infrastructure logic**.

Operating systems differences are influenced by the version of the installed application (that might support different configuration options), how it has been packaged, and where files are placed. Even a configuration file as common as `/etc/ssh/sshd_config` can change in a relevant way on a different OS.

Infrastructure logic may affect in various ways the content of a file. Various factors strictly related to our infrastructure can determine the values of single parameters, for example we can have servers with the very same role that need to use different servers or domains according to the region or the datacenter where they are located. Also, two nodes running the same software may need a completely different configuration depending on their role, as with Puppet configuration in the case of servers and clients.

According to how the configuration files of an application change on our servers we can use different approaches to how we manage them with Puppet:

- All the **configuration files are the same**. With a simple case, we can provide the same file either via the `source` argument or with `content` based on an ERB template which does not have variables interpolation.
- Configuration files are mostly the same, they **change only for a few parameters**. This is a typical case where we can use ERB templates that interpolate variables which might be set via parameters in the class that provides the template. An important thing to consider here is how these parameters change in our nodes, according to what logic (server's role/ zone/ datacenter/ country/ operational environment?) and if this logic may be expressed by what we defined as **nodes identifying variables** in *Chapter 4, Designing Puppet Architectures*. This is important because it gives us glimpses of how correct we have been in setting them.
- Configuration files are **quite different** and one or a few parameters is not enough to express all the differences that occur on our nodes. Here things get trickier. We can use completely different ERB templates (or static files) for each major configuration layout, or we might be tempted to manage these differences with an `if` statement inside a unique template. The latter is a solution I'm not particularly fond of, since, as a general rule, I would avoid placing logic inside files (in this case ERB templates). But it might make sense in cases where we want to avoid excessive duplication of files, which differ just for some portions. In some cases we might prefer a **settings-based** approach, instead of managing the whole configuration file, we can just set the single lines that we need. I find this useful in two main cases:
 - When different OS have quite different configuration files and we just need to enable a specific configuration setting.
 - When we manage with Puppet configuration files web applications that might not be managed via native packages (that generally don't overwrite existing configuration files and are inherently conservative on applications versions). A typical example here can be the database connection settings of, for example, a PHP application.

Applying changes

So we are finally ready to run our Puppet code and apply it to our servers.

Some of the precautions to take here can be applied whenever we work with Puppet. Some are more specific to cases where Puppet is introduced on existing systems and it changes running configurations.

The procedure to follow when we have to apply new Puppet code can definitely vary according to the amount of the managed node, their importance for our business, and the maintenance rules we follow in our company.

Still, a few common points can definitely be recommended:

- **Communicate:** This is a primary rule that should be applied every time. And in these cases it makes particular sense: we are going to change the way some configurations are managed on our systems. Whoever might have the permissions and tasks to modify them should be aware that what was managed before in a manual or some other way, now is managed by Puppet. At least in the operations teams the whole Puppet rollout process should be shared.
- **Test with noop:** There's a wonderfully useful option of the puppet command: `--noop`, it allows us to execute a Puppet run and see its effects without actually making any change to the system. This is extremely useful to see how configuration files will change and spot potential problems before they occur. If we have Puppet running continuously on our servers at regular intervals (being triggered via cron or running as an agent) we should test our code in a dedicated environment before pushing it in the production one. We will review in *Chapter 8, Code Workflow Management*, how to manage the code workflow.
- **Test on different systems:** If our configuration changes according to the OS of a system or its role or any other factor changes we should verify how it behaves for each different case, before propagating the change to all the servers. Also in this case we can use Puppet environments to test our code before pushing it to production even if this is generally not recommended. If we feel confident, we don't have to test each possible case.
- **Propagate the change:** Once we have tested with `--noop` if it doesn't cause harm and once we've applied it to different kinds of servers and we have the expected results, we can be more confident that our change won't disrupt our operations and we can deploy it to production. Propagation of the change may depend on different factors, like the way a Puppet run is triggered, its interval, the topology of our infrastructure and the possibility to roll out changes in a segmented way.

- **Watch reports:** During the deployment of our change it's quite important to keep an eye on Puppet reports and spot errors or unexpected changes. Puppet has a multitude of different report options. Web frontends like The Foreman, Puppet board or Puppet Enterprise provide easy to access and useful reporting features that we can check to verify how our rollout is proceeding.
- **Don't be surprised by skeletons in the closet:** When we apply a configuration on many servers that have been managed manually for many years we are making a major cleanup and standardization activity. We can easily spot old and forgotten parameters that may be incorrect or suboptimal which might refer to old systems which are no longer in production or obsolete settings. This is normal and is a beneficial side effect of the introduction of Puppet.
- **Review and patch uncovered configurations:** It may happen that the configuration provided via Puppet doesn't honor some special case that we hadn't considered. No need to panic. We can fix things in a pragmatic and methodical way, giving priority to the urgent cases (eventually rolling back manually the configuration changed by Puppet and disabling its execution run on the involved servers until we fix it). Often in such cases the exception can be managed on a per-node basis, and for this reason it's useful to have on Hiera hierarchies, or anywhere in our code, the possibility to have configurations to manage special cases per specific nodes.

Things change

Once we introduce a tool like Puppet in our infrastructure, **everything changes** and we should be well aware of this.

There's a wonderful term that describes what configuration management software like Puppet or Chef involve: **Infrastructure as code**; we define our IT infrastructure with formal code, the configurations of our servers, the procedures to set them up, whatever is needed to turn a piece of bare metal or a blank VM in to a system that provides services for our purposes.

When we can use a programming language to configure our systems, a lot of powerful collateral effects take place:

- **Code can be versioned with an SCM:** The history of our commits reflects the history of our infrastructure: we can know who made a change, when and why. There's a huge intrinsic value in this; the power of contextual documentation and communication. Puppet code is inherently the documentation of the infrastructure, and the commits log reflects how this code has been deployed and evolved. It quickly communicates in a single place the rationale and reasons behind some changes much better than sparse e-mail, a wiki, phone calls, or direct voice requests. Also for this reason it is quite important to be disciplined and exhaustive when doing our commits. Each commit on the Puppet code base should reflect a single change and explain as much as possible about it, possibly referring to relevant ticket numbers.
- **Code can be run as many times as wanted:** We have already underlined this concept in *Chapter 1, Puppet Essentials* of this book, but it's worth further attention. Once we express with code how to setup our systems we can be confident that what works once works always: the setup procedure can be repeated and it always delivers the same result (given we have correctly managed resources dependencies). There's a wonderful feeling we gain once we have Puppet on our systems: we are confident that the setup is coherent and is done how we expect it. A new server is installed and all the configurations we expect from it are delivered in a quick and automated way: there is no real need to double check if some settings are correct, if all the common configurations are applied as expected: in a mature and well tested Puppet infrastructure they are. There's more, though, a lot more: ideally we just need our Puppet codebase and a backup of our application data to rebuild from scratch, in reasonable times, our whole infrastructure. In a perfect world we can create a **disaster recovery** site from scratch in a few minutes or hours, given that we have quick and automated deployment procedures (in the cloud this is not difficult), 100% Puppet coverage of all needed configurations, and automated or semi-automated application deployments and restore facilities of our data. A completely automated disaster recovery site setup from scratch is probably not even needed. (It's much more probable that during a crisis we have all our sys admin working actively on an existing disaster recovery site). But whatever can be automated, can save time and human errors during the most critical hours.

- **Code can be tested:** We will review in the next chapter the testing methodologies we have at our disposal when writing Puppet code. However, here it is important to stress that the possibility to test our code allows us to preventively verify how our changes may affect the systems where we want to apply them and make them much more controlled and predictable. This is crucial for tools like Puppet that directly affect how systems operate and can introduce changes (and failures) at scale. We will soon realize that automating the testing of our Puppet changes is possible and recommended and can help us in being more confident on how to deliver our code to production.
- **Code can be refactored and refined:** The more we develop Puppet code the more we understand Puppet and how to make it fit better to our infrastructure. This is a learning process that inevitably has to be done by ourselves and on our infrastructure. We will definitely consider what we wrote a year or just few months ago inaccurate and inappropriate for new the technical requirements that have emerged or are just badly written. The good news is that, as with any type of code, this can be refactored and made better. How and what to fix in our code base should depend on how it affects our work: bad looking code that works and fulfills its duties has less refactoring priority than code that creates problems, is difficult to maintain or doesn't fit new requirements.

It should be clear by now that such a radical change of how systems are managed involves also a change in how the system administrator works.

The primary and most important point is that the sys admin has more and more code to cope with code. This involves that somehow he has to adopt a developer's approach to his work: using a SCM, testing and refactoring code might be activities that are not familiar to a sys admin but have to be tackled and embraced because they are simply needed.

The second radical change involves how systems are managed. Ideally, in a Puppet managed infrastructure we don't even need to SSH to a system and issue commands from the shell: any manual change on the system should be definitely avoided.

This might be frustrating and irritating, since we can quickly do things manually that with Puppet requires a lot more time (write the code, commit, test, apply. A trivial change you can make in five seconds by hand may take you minutes).

The point is that what is done on Puppet remains. It can be applied when the system is reinstalled and is inherently documented: a quick manual fix is easily and quickly forgotten and may make us lose a lot of time when there's the need to replicate it and there's no trace of how, where, and why it was done.

Many of the most annoying and dangerous issues people experience when Puppet is introduced are actually due to the lack of real embracement of the mind set and the techniques that the tools require: people make manual changes and don't implement them in manifests. Then, at the first Puppet run, they are reverted and for any problems that take place Puppet gets the blame. In other cases, there's the temptation to simply disable it and leave the system modified manually. This may make sense in exceptional emergency cases, but after the manual fix the change has to be integrated and Puppet should be re-enabled.

If this doesn't happen we will soon fill our infrastructure with servers where Puppet is disabled and the drift for the configured baselines gets wider and wider, making it more and more difficult, risky and time consuming to re-enable the service.

Puppet friendly infrastructure

We should have realized by now that there are things that Puppet does well and things that are not properly in its chords. Puppet is great at managing files, packages and services, and is less effective in executing plain commands especially if they happen occasionally.

Once we start to introduce Puppet in our infrastructures we should start, where possible, to make things in a way that favors its implementation.

Some time ago in a conference open space I remember a discussion with a developer about how to make a Java application more manageable with Puppet. From his point of view a good API could be a solid approach to make it more configurable. All the systems administrators, and Puppet users, replied that actually that was far from being a nice thing to have: it is much better to have good old plain configuration files.

If we can express it with a file, we can easily manage it with Puppet.

Whatever requires the execution of commands, even if doable, is always considered with skepticism. Installation of software should definitely be done via the OS native packages. Even if most of us have done our definitions that fetch a tar ball, unpack it and compile it, nobody really likes it: that involves a procedural and not a declarative approach, it makes idempotence more difficult and is definitely worse to express with Puppet DSL.

Services should be managed following the OS native approach, be that init, systemd, upstart, or whatever. Managing them with the execution of a command in a custom startup script is definitely not Puppet friendly.

Configuration files from common stacks of applications should be leveraged. For example when we have to manage the Java settings on an application server it makes sense to manage them always in the same place; standardizing them is a matter of overall coherence and easier management.

Once we gain affinity and expertise with Puppet we quickly understand what are the things that can be done quickly and safely and what are the things that make our life harder, requiring more resources or quick and dirty patches.

It is our responsibility to discuss this with the developers and the stakeholders of an application to find a collaborative approach to its setup that makes it easier to be managed with Puppet while preserving the requested implementation needs.

Puppet-friendly software

During this chapter we have seen some characteristics that make working with Puppet a much better experience. Let's remark on some of these characteristics with insights about how its deficiencies can make our code more complex or less likely to behave in a deterministic way. Some of these characteristics depend on how the software is packaged, but others may depend on the software itself. Let's look at them:

- Use the **standard packaging system** of the OS used in your infrastructure. This way the `package` resource type can be used in a homogeneous way. If the software to be installed is not officially packaged for the needed packaging system it's important to evaluate the possibility of packaging it. This will allow you to simplify the implementation of its Puppet module. The alternative could probably be to use some custom scripts possibly wrapped around `defines` to call them in their use cases, something that adds complexity to the deployment.
- Use **composable configuration files**. As seen in the previous section, should prefer good old plain text configuration files are preferred over other more complex systems based on APIs or databases. Puppet has very good support for deploying files based on static sources or templates, which makes it easy to deploy configuration. Sometimes the configuration of an application or service is too complex to be managed by only a template or static file. In these cases, we should expect this software to support composable configuration files by reading them from a `conf.d` directory. Then our Puppet modules will contain the logic to decide what files to use in each role while keeping the files as simple as possible.

- **Install services disabled by default.** When installing packages from official distributions services can be enabled and started by default, this can lead to unexpected behaviors. For example the installation can fail if the service starts by default in an already used port, or a package that installs multiple services could start one that we don't want running on our system. These errors will make puppet execution fail, leaving the node in an unknown state. It can also hide future problems as the code may correctly apply in a working system, but it will fail in a new node. Sometimes these problems can be worked around by deploying the configuration before installing the package, but then problems with users or permissions can arise. The ideal workflow for installing services with Puppet would be to install with `package`, configure with `files`, and enable and/or start with `service`.
- Something that is sometimes complex to implement, but is very desirable, is **online configuration reload**. As we are seeing throughout this book, Puppet is in charge of keeping our infrastructure in an expected state. This implies that sooner or later we'll want to change the configuration of our services. One of the most used approaches is to modify a `file` and subscribe the `service` to it that will provoke a restart of the service. Sometimes this is not acceptable, as the service will drop connections or lose data. There are a couple of options to manage this scenario. One is to coordinate the restarts, but Puppet alone is not very good at doing this kind of coordination. The other option is to check if the software is able to receive signals requesting configuration reloads and use the `hasreload` parameter of service resources. This last option is preferable when working with Puppet because it fits perfectly in the usual workflows, but the software has to support it.
- Another desirable characteristic of Puppet-friendly software is **client-side registration**. When a service needs to know a collection of clients (or workers, backend), it uses to be easier to manage if the clients are the ones taking the initiative to join the cluster. Puppet also has support for the opposite scenario, but it uses to imply the use of exported resources and the need to reload the configuration in the service, which is not so straightforward.

In most of the cases we cannot take the decision to use software just because it's more Puppet friendly. But if we are managing our infrastructure with Puppet it's worth considering these characteristics. If we can choose between different options, we can add these arguments to the balance of the software that implements them. If we cannot choose, check if the software to be used can be used in a Puppet-friendly way even if it's not its default behavior.

Summary

The introduction of Puppet on an infrastructure is a long and intriguing voyage without return. It requires planning, patience, method, experience, and skills, but it brings huge results.

There's definitely not a unique way to face it. In this chapter, we have exposed the general scenarios we might face, the possible alternative approaches, and have suggested a step-by-step procedure articulated in different phases: information gathering, priority setting, decision making, code development, application to production, and testing.

These phases should be reiterated at each step, with more or less emphasis on what matters to get things done.

We have also faced the changes that such a process involves: how we need a new mindset and new processes to sustain a Puppet setup.

Complementary to this is an effective management of our code: how it's versioned, reviewed, tested, and delivered to production. These are some of the topics we are going to discuss in the next chapter.

8

Code Workflow Management

All the Puppet manifests, public shared modules, site modules, and (Hiera) data are the code and data we create. We need tools and workflows to manage them.

In this chapter, we are going to review the existing tools and techniques to manage Puppet's code workflow, from when it is written to when it is deployed to production.

Most of the people in the Puppet world use Git to version their code, so we will refer mostly to Git, but similar processes can be followed if we manage our code with subversion, mercurial, or any other source code management tool.

In this chapter, we give an overview of the tools that can help us with our Puppet code. We will cover the following topics:

- Write with **Geppetto** and **Vim**
- Manage with **Git**
- Review with **Gerrit**
- Test modules with **rspec-puppet**
- Test Puppet runs with **Beaker** and **Vagrant**
- Deploy with **librarian-puppet** or **r10k**
- Automate with **Travis** or **Jenkins**

Write Puppet code

Each of us has a favorite tool for code writing. It may change according to the language we are using, our familiarity with the software or a preference of the moment.

Whatever tool we use, it should make our experience as smooth, productive, and enjoyable as possible.

I am a **Vim** guy, without being a guru. Having a system admin background, grown with bread and shell, I am comfortable with the possibility of using the same tool, *wherever* I am, on the local terminal or the remote SSH session: more or less we can expect to find Vim on any system under our keyboard.

A developer, I guess, may feel more comfortable with a tool that runs on his computer and can greatly enhance the writing experience, with syntax checks, cross references, and all the power of an IDE.

For this, there is **Geppetto**, a full-featured IDE, based on **Eclipse** and dedicated to Puppet code. Other popular editors also have Puppet plugins that can be quite useful.

The good news is that all of them can make life more enjoyable and productive when we write Puppet code.

Geppetto

Geppetto (<http://puppetlabs.github.io/geppetto/>) is one of the tools of reference for writing Puppet code. It is an open source software by Puppet Labs, which is a result of the acquisition of the startup Cloud smith that developed it.

We can install Geppetto as a standalone application or as an Eclipse plugin, if we are already using it.

It has very useful features such as:

- Code syntax and style checking
- Contextual documentation
- Integration with the Puppet Forge
- Integration with PuppetDB
- All the features inherited from Eclipse

When we launch it, we are prompted for a directory from which to create our Geppetto **workplace**. In a workplace, we can create projects that most of the time are modules we can directly import from the Forge or a SCM repository.



Geppetto may be memory hungry with large projects; if we bump into Internal Error: Java heap space, we probably have to enlarge the memory pools for the JVM Geppetto runs into. In `geppetto.ini` (or `eclipse.ini` if we use Geppetto as a plugin), we can set something like this:

```
-XX:MaxPermSize=256m
-vmargs
-Xms384m
-Xmx512m
```

Vim

If we have to think about an evergreen tool that has accompanied many sys admins' keyboards for many years, I think that Vim is one of the first names that comes to mind.

Its power and flexibility is also well expressed in how it can be made to manage Puppet code effectively.

There are two Vim bundles that are particularly useful for us:

- **vim-puppet** (<https://github.com/rodjek/vim-puppet>): This is a syntax highlighter and formatting tool, written by Tim Sharpe, well known in the Puppet community for being the author of hugely popular tools such as **librarian-puppet**, **puppet-lint**, and **rspec-puppet** (all of which are going to be discussed in this chapter).
- **Syntastic**: This is a syntax-checking tool that works for many popular languages and has the capability to check the style and syntax of our manifests right when we save them.

To install them easily we need to install **Pathogen** (a Vim extension that allows easy management of plugins) by copying the `~/.vim/autoload/pathogen.vim` file from <https://raw.githubusercontent.com/tpope/vim-pathogen/master/autoload/pathogen.vim>.

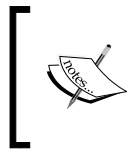
Then, be sure to have Pathogen installed in the `~/.vimrc` of our home directory:

```
call pathogen#infect()
syntax on
filetype indent plugin on
```

Once we have Pathogen loaded, we can easily add bundles in the `.vim/bundle` directory:

```
git clone git://github.com/rojdjek/vim-puppet.git .vim/bundle/puppet
git clone git://github.com/scrooloose/syntastic.git .vim/bundle/
syntastic
```

Now, we can use Vim with these powerful plugins that make coding with Puppet DSL definitively more comfortable.



If, after these changes, we have problems when pasting multiple lines of text from the clipboard (each new line has one more indent level and # signs at the beginning), try to issue the Vim command `:set paste`.

Git workflows

If we want to work and prosper with Git, we have to firmly grasp its principles and the main workflow approaches. There is much theory and some alternatives on how we can manage our Puppet code in a safe and comfortable way using Git.

In this section, we will review:

- The Git basic principles and commands
- Some useful Git hooks

Code management using Git

Git is generally available as the native package in every modern OS. Once we have installed it, we can configure our name and e-mail (that will appear in all our commits) with:

```
git config --global user.name "Alessandro Franceschi"
git config --global user.email al@lab42.it
```

These commands simply create the relevant entries in the `~/.gitconfig` file. We can add more configurations either by directly editing this file or with the `git config` command, and check their current values with `git config -l`.

To create a Git repository, we just have to move into the directory that we want to track and simply type `git init`. This command initializes a repository and creates a local `.git` directory where all Git data is stored.

Now we can type `git status` to see the status of the files in our repository. If we have files in this directory, we will see them as untracked. This means that they are not managed under Git and, before being able to commit changes on our files, we have to stage them; this is done by the `git add` command, which adds all the files from the current working directory to the index (we can add only specific and selected files or changes). If we type `git status` again, we will notice that the files we added are ready to be committed, we can use `git commit` to create a commit with all the changes in the files we've previously staged. Our default editor is open and we can type the title (first line) and the description of our commit.

For later readability and better management, it is important to make single and atomic commits that involve only the changes for a specific fix, feature, or ticket.

Now we can type `git log` to see the history of our commits on this repository.

Git is a distributed SCM; we can work locally on a repository, which is a clone of a remote one. To clone an existing repository, we can use the `git clone` command (we have seen earlier, with Vim bundles, some usage samples).

Once we have cloned a repository, we can use `git push` to update it with our local commits and `git pull` to retrieve all the changes (commits) made there since we cloned it (or made our latest pull).

We may want to work on separated branches where we can manage different versions of our code. To create a new branch, we use `git branch <branchname>`, to work inside a branch we type `git checkout <branchname>`. When we are inside a branch, all the changes we make and commit are limited and confined to that branch; if we switch branches, we will see our changes to the local files magically disappearing. To incorporate the commits made in a branch into another branch, we can use the `git merge` or `git rebase` commands.

These are just the very basics of Git, which just shows the kind of commands we might end up using when working with it, if we want to learn more, there are some great resources online:

- <http://try.github.io>: This is an interactive tutorial to learn Git's basics
- <http://gitready.com>: This is a site full of hints and documents

Git hooks

Git hooks are scripts that are executed when specific Git actions are done. We can add a hook just by placing an executable file under our local `.git/hooks` directory.

The name of the file reflects the phase when the hook is executed.

When we deal with Puppet, we have generally three different hooks that we can use:

- `git/hooks/pre-commit`: This is executed before finalizing the commit. Here, it makes sense to place syntax and lint checks that prevent us from committing code with errors.
- `git/hooks/pre-receive`: This is executed before accepting `git push`, it may be placed on our central Git repository, and can be quite similar to the `pre-commit` one. The difference here is that the checks are done on the central repository when changes already committed on remote working repositories are pushed.
- `git/hooks/post-receive`: This is executed on the central repository server after push. It can be used to automatically distribute our committed code to a testing (or production) Puppet environment.

Usage possibilities for hooks are endless; you can find some useful ones at <https://github.com/drwahl/puppet-Git-hooks>.

Environments and branches

When we want to manage our Puppet code with Git, we can follow different approaches. Our main objectives are:

- To be able to version our code
- To be able to distribute it
- To be able to test our code before pushing it into production

The first two points are implicit with Git usage; the third one can be achieved using Puppet's environments. Since we can set different local directories for the `modulepath` and the `manifest` that are mapped to different Puppet environments on the Puppet Master, we can couple different Git branches or working repositories to them.

Here, we outline two possible approaches:

- The first is the officially recommended one, which is based on **automatic environments** and a well-established Git workflow where a new branch is created for each new feature or bug fix. Here, we have to branch, merge, pull, and push from different branches.
- The second one is a **simplified approach**, possibly easier to manage for Git beginners, where we mostly pull and push always from the same branch.

Branch based automatic environments

In the first approach, since we may have an unpredictable number of branches, which are created and destroyed regularly, it's used the possibility to create automatic environments with Puppet, which involve a configuration on `puppet.conf` such as the following:

```
[main]
  server = puppet.example.com
  environment = production
  confdir = /etc/puppet
[master]
  environment = production
  manifest = $confdir/environments/$environment/manifests/site.pp
  modulepath = $confdir/environments/$environment/modules
```

This is coupled with the presence of a `post-receive` Git hook that automatically creates the relevant environment directories. Check out <http://puppetlabs.com/blog/git-workflow-and-puppet-environments> for details.

Whoever writes Puppet code can work on any change in a separate Git branch, which can be tested on any client specifying the environment.

Passage to production is generally done by merging the separated branches in the master one.

Simplified developer workdir environments

If our team has limited affinity with Git, we might find it easier to manage changes in a different, simpler, way — people always work on the master branch, they have a local working copy of the Puppet code where they work, test and commit. Code can be promoted to review for production with a simple `git push`.

Also, in this case, it is possible to map in `puppet.conf` the Puppet Master environments to the local working directories of the developers, so that they can test their code on real servers before pushing it to production.

We may evaluate variations on the described alternatives, which can adapt to our business requirements, team's skills, and preferred workflows. In any case, the possibility of testing Puppet code before committing it is important, and a good way to do that is to use Puppet environments that map to the directories where this code is written.

Code review

Reviewing the code before including it in any of our production branches is a good practice due to the great advantages it provides. It helps to detect problems in the code as early as possible. Some things that are out of the test's scope as code readability or variable naming can be detected by other members of the team.

When there's a team of people who work collaboratively on Puppet, it is quite important to have good communication among its members and a common vision on how the code is organized, where logic and data are placed and what are the principles behind some design decisions.

Many mistakes while working on Puppet are due to incomplete knowledge of the area of effect of code changes, and this is due, most of the time, to bad communication.

For this reason, any tool that can boost communication, peer review, and discussion about code can definitively help in having a saner development environment.

Gerrit

When we work with Git, the natural companion to manage peer review and workflow authorization schemes is **Gerrit**, a web interface made by Google for the development of Android, that integrates perfectly with Git and allows commenting on any commit, allows you to vote for them, and has users that may authorize their acceptance.

Different user roles with different permission schemes and authorization workflows can be defined. Once new repositories are added, they can be cloned via Git:

```
git clone ssh://al@gerrit.example.com:29418/puppet-modules
```

When we work on repositories managed under Gerrit, we have to configure our Git to push to the special `for` ref, instead of the default `heads`. This is the key step that allows us to push our commits to a special intermediary place where they can be accepted or rejected:

```
git config remote.origin.push refs/heads/*:refs/for/*
```

We also need a precommit hook that automatically places a (required) **Change-ID** in our commits:

```
gitdir=$(git rev-parse --git-dir) ; scp -p -P 29418 \
al@gerrit.example.com:hooks/commit-msg ${gitdir}/hooks/
```

Once we have made our setup, we can normally work on our local repo with Git, and when we push our changes they are submitted to Gerrit, to the special `refs/for` for code peer review. When a commit is accepted, Gerrit moves it to `refs/heads` from where it can be pulled and distributed.

Gerrit can be introduced at any time in our workflow to manage the acceptance of commits in a centrally managed way and, besides the client's configurations, we have seen it doesn't require further changes in our architecture.



A great tutorial on Gerrit can be seen here: <http://www.vogella.com/tutorials/Gerrit/article.html>

Online resources for peer review

There are various online services that can be used for peer review and provide an alternative to an on-premise solution based on Gerrit.

GitEnterprise (<https://gitent-scm.com>) offers a Gerrit and code repository online service.

RBCommons (<https://rbcommons.com>) is the SaaS version of the review board's (<http://www.reviewboard.org/>) open source code review software.

Repository management sites such as GitHub (<https://github.com>) or BitBucket (<https://bitbucket.com>) offer, via the web interface, an easy way to fork a repository, make local changes and push them back to the upstream repository with pull requests that can be discussed and approved by the upstream repository owner.

Testing Puppet code

It has been clear for years that there is the strong need to be able to test how changes to our Puppet code can affect our infrastructure.

The topic is quite large, complex, and, to be honest, not completely solved, but there are tools and methods than can help us in safely working with Puppet in a production environment.

We can test our code with these tools:

- The command `puppet parser validate`, to check the syntax of our manifests
- **puppet-lint** (<http://puppet-lint.com/>) to check that the style of our code conforms with the recommended style guide

- **rspec-puppet** to test the catalog and the logic of our modules
- **rspec-puppet-system** and **Beaker**, to test what happens when our catalog is applied to a real system

We can also follow some procedures and techniques, such as:

- Using the `--noop` option to verify what would be the changes before applying them
- Using Puppet **environments** to try our code on test systems before pushing it into production
- Having canary **nodes** where Puppet is run and changes are verified
- Having a gradual, clustered, **deployment rollout** procedure

Using rspec-puppet

Rspec-puppet (<http://rspec-puppet.com>) makes it possible to test Puppet manifests using *rspec*, a widely used Ruby behavior-driven development framework.

It can be installed as gem:

```
gem install rspec-puppet
```

Once installed, we can move into a directory of a module and execute:

```
rspec-puppet-init
```

This command creates a basic *rspec* environment composed by:

- The `Rakefile`, in our module's main directory, where the tasks we can run with the tool **rake** and which can trigger the execution of our tests are defined.
- The `spec/` directory that contains all the files needed to define our tests; `spec/spec_helper.rb` is a small Ruby script that helps in setting up Puppet's running environment during the tests' execution; `spec/classes`, `spec/defines`, and `spec/functions` are subdirectories where the tests for classes, defines, and functions should be placed.
- The `spec/fixtures` directory is used, during testing, to temporarily copy the Puppet code we need to fulfill our tests. It's possible to define in a `fixtures.yml` file, eventual other dependency modules to fetch and use during the tests.

Test files have names that reflect what they are testing, for example, if our module is called `apache`, the tests for the `apache` class could be placed in `spec/classes/apache_spec.rb` and the tests of a define called `apache::vhost` are in `spec/defines/apache_vhost_spec.rb`.

In these files, we have the normal Ruby spec code, where the test condition and the expected result is defined, in terms of Puppet resources and their properties.

This is a sample test file for an `apache` class (taken from Puppet Labs' `apache` module).

The first line requires the `spec_helper.rb` file previously described as:

```
require 'spec_helper'
```

Then, a Debian context is defined for which the relevant facts are provided:

```
describe 'apache', :type => :class do
  context "on a Debian OS" do
    let :facts do
      let :facts do
        {
          :id                        => 'root',
          :kernel                   => 'Linux',
          :lsbdistcodename          => 'squeeze',
          :osfamily                 => 'Debian',
          :operatingsystem          => 'Debian',
          :operatingsystemrelease  => '8',
          :path                     => '/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin',
          :concat_basedir           => '/dne',
          :is_pe                    => false,
        }
      end
    end
  end
end
```

Under this context (defined according to the specified facts or parameters) are then listed the resources that we expect in a catalog; note that it is possible to use the `is_expected.tocontain_*` matcher for any Puppet resource type, and each argument of that resource can be tested:

```
it { is_expected.tocontain_package("httpd").with(
  'notify' => 'Class[Apache::Service]',
  'ensure' => 'installed'
) }
it { should contain_user("www-data") }
```

We can also set specific parameters and test the behavior we expect when they are provided:

```
describe "Don't create user resource" do
  context "when parameter manage_user is false" do
    let :params do
      { :manage_user => false }
    end

    it { is_expected.not_to contain_user('www-data') }
    it { is_expected.to contain_file("/etc/apache2/apache2.conf") }
    with_content %r{^User www-data\n} }
  end
end
```

Note the `with_content` matcher that permits to verify the same contents of a file.

Rspec-puppet's intended usage is to verify that the logic of our module is correctly applied under different conditions, for example, testing its behavior when different parameters are provided or when different OSes are simulated. It is useful to prevent regressions during code refactoring and to quickly validate pull requests from contributors.

It is not intended to validate the effect of our code when applied on a system.

This is a key point to understand: rspec-puppet works on the catalog and checks whether Puppet resources are correctly provided. It does not test what happens when those resources are actually applied for real: in software testing terms, rspec-puppet is a unit-testing tool for modules.

Vagrant

Vagrant (<http://www.vagrantup.com>) is a very popular tool created by Mitchell Hashimoto to quickly create **Virtual Machines (VM)** mostly for testing and developing purposes.

One of Vagrant's strong points is the ability to run provisioning scripts when a VM is created and, among the others, there's the possibility to directly run Puppet either with the `apply` command, pointing to a local manifest, or with `agent`, pointing to a Puppet Master.

This makes Vagrant a perfect tool to automate the testing of Puppet code and to replicate in a local environment development and test setups that can be aligned to the production ones.

Once we have Vagrant and VirtualBox installed (its virtualization technology of reference, even if plugins are available to manage other VM backends), we just need to create `Vagrantfile` where its configurations are defined.

We can run Vagrant without arguments to see the list of available actions. The most common ones are:

- `vagrant status`: This shows the current VM's status.
- `vagrant up [machine]`: This turns on all the existing VMs or the specified one. The VM is generated from a base box. If that box is not locally stored, then it's downloaded and saved in `~/.vagrant.d/boxes/`.
- `vagrant provision [machine]`: This runs the provisioning scripts on all or one VM.
- `vagrant halt [machine]`: This stops all or one specific VM.
- `vagrant destroy [machine]`: This destroys all or one specific VM (their content will be erased, they can then be recreated with `vagrant up`).
- `vagrant ssh <machine>`: SSH into a running VM. Once logged, we should be able to run `sudo -s` to gain root privileges without entering a password.

Puppet can be used as a provisioner for the virtual machines started with Vagrant; for that we can add these lines to the `vagrant` file:

```
config.vm.provision :puppet do |puppet|
  puppet.hiera_config_path = 'hiera-vagrant.yaml'
  puppet.manifests_path = 'manifests'
  puppet.manifest_file = 'site.pp'
  puppet.module_path = [ 'modules' , 'site' ]
  puppet.options = [
    '--verbose',
    '--report',
    '--show_diff',
    '--pluginsync',
    '--summarize',
  ]
end
```

With this configuration, we can place the Puppet code that will be executed on virtual machine provisioning. Modules in the `modules` and `site` directories, the first manifest file applied to nodes is `manifests/site.pp`, the Hieradata configuration file is `hiera-vagrant.yaml`, its data is in `hieradata`.

The ability to quickly and easily create a VM to run Puppet makes Vagrant a perfect tool to test the effects of our changes on Puppet manifests in automated or manual ways on a real system.

Beaker and beaker-rspec

Beaker (<https://github.com/puppetlabs/beaker>) has been developed internally at Puppet Labs to manage acceptance tests for their products under these principles:

- Tests are done on virtual machines that are created and destroyed on the fly
- Tests are described with rspec style syntax
- Tests also define the expected results on a system where the tested code is applied

Beaker tests are executed on one or more **System Under Test (SUT)**, which may be run under multiple IaaS, hypervisors, or container executors. Beaker-rspec (<https://github.com/puppetlabs/beaker-rspec>), also developed by Puppet Labs, integrates beaker with rspec, allowing rspec to execute tests in hosts provisioned by beaker.

They can be installed using `gem`:

```
gem install beaker beaker-rspec
```

Tests are configured in the `spec/acceptance` directory of a module using one or more hosts, which may have different roles and different OSes. Their configuration is defined in YAML files generally placed in the `spec/acceptance/nodesets` directory.

Virtual machines or containers in which we want to run the tests have to be configured in the host's file; by default, beaker-rspec needs one in `spec/acceptance/nodesets/default.yml`, for example, with this file, tests would be run using Vagrant in Ubuntu 14.04:

```
HOSTS:
  ubuntu-1404-x64:
    roles:
      - master
      - agent
      - dashboard
      - cloudpro
    platform: ubuntu-1404-x86_64
    hypervisor: vagrant
    box: puppetlabs/ubuntu-14.04-64-nocm
```

```

    box_url: https://vagrantcloud.com/puppetlabs/boxes/ubuntu-14.04-
64-nocm
CONFIG:
  nfs_server: none
  consoleport: 443

```

Tests have to use `rspec` and `beaker` DSLs, as a minimal example, we can have a helper that installs puppet in a host provided by beaker, and a test that just checks that puppets runs correctly and a file is created.

The helper can be placed in `spec/spec_helper_acceptance.rb` and would look like this:

```

require 'beaker-rspec'
hosts.each do |host|
  install_puppet
end

```

If we need to install some additional modules such as `stdlib`, we could add this code to the file:

```

c.before :suite do
  RSpec.configure do |c|
    hosts.each do |host|
      on(host, puppet('module', 'install', 'puppetlabs-stdlib'))
    end
  end
end

```

If what we are testing is a module, we are developing, we may also install it, adding this code inside the same block where `stdlib` is installed:

```

install_dev_puppet_module_on(host,
  :source => File.expand_path(
    File.join(File.dirname(__FILE__), '..'),
  :module_name => 'example',
  :target_module_path => '/etc/puppet/modules')

```

Once our helper is ready, we can start to define tests, to better understand their structure, look at this dummy example that we can place in `spec/acceptance/example_spec.rb`:

```

require 'spec_helper_acceptance'

describe 'example class' do
  let(:manifest) {
    <<-EOS

```

```
    file { ['/example-test': ensure => 'present' }
    EOS
  }

  it 'should run without errors' do
    result = apply_manifest(manifest)
    expect(@result.exit_code).to eq 0
  end

  describe file('/example-test') do
    it { is_expected.to exist }
  end
end
```

First of all, we include our helper, so we are sure that Puppet or any other dependency is installed, then we describe our suite, that is divided in three parts:

1. With `let` we define any variable or state needed for the tests; in this case, we define the manifest to run with Puppet in which we just create a file.
2. Then, we run Puppet as a step that should execute without errors.
3. Finally, we describe the expected state of the file created.

If the Puppet run fails or if the condition is not met, `beaker-rspec` will report the error.

To execute the tests, we use `rspec`:

```
rspec spec/acceptance
```

Following this schema, we can test any puppet code, and in several platforms, this is what provides a powerful tool to ensure the quality of our code.

Deploying Puppet code

Deployment of Puppet code is, most of the times, a matter of updating modules, manifests, and Hiera data on relevant directories of the Puppet Master.

We deal with two different kinds of code which involve different management patterns:

- Our modules, manifests, and data
- The public modules we are using

We can manage them in the following ways:

- Using Git—eventually using Git submodules for each Puppet module
- Using the puppet module, for the public modules published on the Forge
- Using tools such as `librarian-puppet` and `r10k`
- Using other tools or custom procedures we might write specifically for our needs

Using librarian-puppet for deployments

Librarian-puppet (<http://librarian-puppet.com>) has been developed to manage the installation of a set of modules from the Puppet Forge or any Git repository. It is based on `Puppetfile` where the modules and the versions to be installed are defined:

```
forge "http://forge.puppetlabs.com"

# Install a module from the Forge
mod 'puppetlabs/concat'

# Install a specific version of a module from the Forge
mod 'puppetlabs/stdlib', '2.5.1'

# Install a module from a Git repository
mod 'mysql',
  :git => 'git://github.com/example42/puppet-mysql.git',

# Install a specific tag of a module from a Git repository
mod 'mysql',
  :git => 'git://github.com/example42/puppet-mysql.git',
  :ref => 'v2.0.2'

# Install a module from a Git repository at a defined branch
mod 'mysql',
  :git => 'git://github.com/example42/puppet-mysql.git',
  :ref => 'origin/develop'
```

In our `modulepath`, we can install all the modules referenced in the local `Puppetfile` using:

```
librarian-puppet install
```


Deploying code with r10k

r10k has been written by Adrien Thebo, who works in Puppet Labs, to manage deployment of Puppet code from Git repositories or the Forge. It fully supports **librarian-puppet**'s format for the `Puppetfile` but is an alternative tool that empowers a workflow based on Puppet's dynamic environments.

It can be installed as gem:

```
gem install r10k
```

In its configuration file, `/etc/r10k.yaml`, we define one or more source Git repositories to deploy in the defined `basedir`:

```
:cachedir: '/var/cache/r10k'
:sources:
:site:
  remote: 'https://github.com/example/puppet-site'
  basedir: '/etc/puppetlabs/code/environments'
```

The interesting thing is that for each branch of our Git source, we find a separated directory inside the defined `basedir`, which can be dynamically mapped to Puppet environments.

To run a deploy from the repository where we have a `Puppetfile` we can run:

```
r10k deploy environments -p
```

This creates a directory for each branch of our repository under `/etc/puppet/environments`, and inside these directories, we find all the modules defined in the `Puppetfile` under the `modules/` directory.

Propagating Puppet changes

Deployment of Puppet code on production is a matter of updating the files on the directories served by the Puppet Master (or, in a Masterless setup, distributing these files on each node), but, contrary to other typical application deployments, the process doesn't end here, we need to run Puppet on our nodes in order to apply the changes.

How this is done largely depends on the policy we follow to manage Puppet execution.

We can manage Puppet runs in different ways and this affects how our changes can be propagated:

- Running Puppet as a service—in this case, any change on the Puppet production environment (or what is configured as default) is propagated to the whole infrastructure in the run interval timeframe.
- Running Puppet via a cron job has a similar behavior; whatever is pushed to production is automatically propagated in the cron interval we defined. Also in this case, if we want to make controlled executions of Puppet on selected servers, the only approach involves the usage of dedicated environments before the code is promoted to the production environment.
- We can trigger Puppet runs in a central way, for example via MCollective (check <http://www.slideshare.net/PuppetLabs/presentation-16281121> for good presentation on how to do it); once our code has been pushed to production, we still have the possibility to manually run it on single machines before propagating it to the whole infrastructure. The complete rollout can then be further controlled either using canary nodes, where changes are applied and monitored first, or, in large installations, having different clusters of nodes where changes can be propagated in a controlled way.

Whatever are the patterns used, it's very important and useful to keep an eye on the Puppet reports, and quickly spot early signs of failures caused by Puppet's runs.

Puppet continuous integration

We have reviewed the tools than can accompany our code from creation to production.

Whatever happens after we commit and eventually approve our code can be automated.

It is basically a matter of executing commands on local or remote systems that use tools like the ones we have seen in this chapter for the various stages of a deployment workflow.

Once we have these single bricks that fulfill a specific function, we can automate the whole workflow with a **Continuous Integration (CI)** tool that can run each step in an unattended way and proceed to the following if there are no errors.

There are various CI tools and services available; we will concentrate on a pair of them, particularly popular in the Puppet community:

- **Travis:** An online CI as a service tool
- **Jenkins:** The well known and widely used Hudson fork

Travis

Travis (<https://travis-ci.org>) is an online Continuous Integration service that perfectly integrates with GitHub.

It can be used to run tests of any kind, in Puppet world, it is generally used to validate the module's code with **rspec-puppet**. Refer to online documentation on how to enable the Travis hooks on GitHub (<http://docs.travis-ci.com/user/getting-started/>); on our repo we manage what to test it with a `.travis.yml` file:

```
language: ruby
rvm:
  - 1.8.7
  - 1.9.3
script:
  - "rake spec SPEC_OPTS='--format documentation'"
env:
  - PUPPET_VERSION="~> 2.6.0"
  - PUPPET_VERSION="~> 2.7.0"
  - PUPPET_VERSION="~> 3.1.0"
matrix:
  exclude:
    - rvm: 1.9.3
      env: PUPPET_VERSION="~> 2.6.0"
      gemfile: .gemfile.travis

gemfile: .gemfile.travis
notifications:
  email:
    - HYPERLINK "mailto:al@lab42.it"al@lab42.it
```

As we can see from the preceding lines, it is possible to test our code on different Ruby versions (managed via **Ruby Version Manager (RVM)** <https://rvm.io>) and different Puppet versions.

It's also possible to exclude some entries from the full matrix of the various combinations (for example, the preceding example executes the `rake spec` command to run puppet-rspec tests in five different environments: 2 Ruby versions * 3 Puppet versions - 1 matrix exclusion).

If we publish our shared Puppet modules on GitHub, Travis is particularly useful to automatically test the contributions we receive, as it is directly integrated on GitHub's pull requests workflow, which is commonly used to submit author fixes or enhancements on the code to a repository (check out <https://help.github.com/categories/63/articles> for details).

Jenkins

Jenkins is by far the most popular open source Continuous Integration tool. We are not going to describe how to install and use it and will just point out useful plugins for our purposes.

A Puppet-related code workflow can follow common patterns; when a change is committed and accepted, Jenkins can trigger the execution of tests of different kinds and, if they pass, can automatically (or after human confirmation) manage the deployment of the Puppet code on production (typically, by updating the directories on the Puppet Master that are used by the production environment).

Among the multitude of Jenkins plugins (<https://wiki.jenkins-ci.org/display/JENKINS/Plugins>) the ones that are most useful for our purposes are:

- **ssh**: This allows execution of a command on a remote server. This can be used to manage deployments with `librarian-puppet` or `r10k` or execute specific tests.
- **RVM/rbenv**: This integrates with RVM or Rbenv to manage execution of tests in a controlled Ruby environment. They can be used for `rspec-puppet` and `puppet-lint` checks.
- **GitHub / Gerrit**: This integrates with GitHub and Gerrit to manage code workflow.
- **Vagrant**: This integrates with Vagrant for tests based on real running machines.

Testing can be done locally on the Jenkins server (using the `rvm/rbenv` plugins) or on any remote host (via the SSH plugin or similar tool); deployment of the code can be done in different ways, which will probably result in the execution of a remote command on the Puppet Master.

Summary

In this chapter, we have reviewed the tools that can help us from when we start to write our Puppet code, to how we manage, test, and deploy it.

We have seen how to enhance the writing experience on **Geppetto**, Puppet Labs' official Puppet IDE and **Vim**, a sysadmins' evergreen tool, how to version and manage code with **Git**, and eventually, how to introduce a peer review and approval system such as **Gerrit**.

We then saw the different tools and methodologies available to test our code: from simple syntax checks, which should be automated in Git hooks; to style checks with **puppet-lint**, from unit testing on modules with **puppet-rspec**; to real life acceptance tests on running (and ephemeral) Virtual Machines, managed with **Vagrant**; and using tools like **Beaker**.

We finally faced how Puppet code can be delivered to production, with tools such as **librarian-puppet** and **r10k**.

The execution of all these single tools can be automated with Continuous Integration tools, either to trigger tests automatically when a Pull Request is done on a GitHub repository, such as **Travis**, or to manage the whole workflow from code commit to production deployment, with **Jenkins**.

Our overall needs are to be able to write Puppet code that can be safely and quickly promoted to production; tools and processes can help people in doing their work at best.

The next chapter is about scaling—how to make our Puppet setup grow with our infrastructure.

9

Scaling Puppet Infrastructures

There is one thing I particularly like about Puppet: its usage patterns can grow with the user's involvement. We can start using it to explore and modify our system with `puppet resource`, we can use it with local manifests to configure our machine with `puppet apply`, and then we can have a central server where a `puppet master` service provides configurations for all our nodes, where we run the `puppet agent` command.

Eventually, our nodes' number may grow, and we may find ourselves with an overwhelmed Puppet Master that needs to scale accordingly.

In this chapter, we review how to make our Master grow with our infrastructure and how to measure and optimize Puppet performances. You will learn the following:

- Optimizing Puppet Master with **Passenger**
- Optimizing Puppet Server based on **Trapperkeeper**
- Horizontally scaling Puppet Masters
- Load balancing alternatives
- Masterless setups
- Store configs with PuppetDB
- Profiling Puppet performances
- Code optimization

Scaling Puppet

Generally, we don't have to care about Puppet Master's performances when we have few nodes to manage.

Few is definitively a relative word; I would say any number lower than one hundred nodes, which varies according to various factors, such as the following:

- **System resources:** The bare performances of the system, physical or virtual, where our Puppet Master is running are, obviously, a decisive point. Particularly needed is the CPU, which is devoured by the `puppet master` process when it compiles the catalogs for its clients and when it makes MD5 checksums of the files served via the fileserver. Memory can be a limit too while disk I/O should generally not be a bottleneck.
- **Average number of resources for node:** More resources we manage in a node, the bigger the catalog becomes, and it takes more time to compile it on Puppet Master, to deliver it via network and finally to receive and process the clients' reports.
- **Number of managed nodes:** The more nodes we have in our infrastructure, the more work is expected from Puppet Master. More precisely, what really matters for the Master is how many catalogs it has to compile per unit of time. So the number of nodes is just a factor of a multiplication, which also involves the next point.
- **Frequency of Puppet runs for each node:** The default 30 minutes, when Puppet runs as a service, may be changed, and has a big impact on the work submitted to the Master.
- **Exported resources:** If we use exported resources, we may have a huge impact on performances, especially if we don't use PuppetDB as a backend for `storeconfigs`.

As simple as `puppet apply`

The simplest way we can use Puppet is via the `apply` command. It is simple but powerful, because with a single `puppet apply`, we can do exactly what a catalog retrieved from the Puppet Master would do on the local node.

The manifest file we may apply can be similar to our `site.pp` on Puppet Master; we just have to specify `modulepath` and eventually, the `hiera_config` parameters will be able to reproduce the same result we would have with a client-server setup:

```
puppet apply --modulepath=/etc/puppetlabs/code/modules:/etc/puppetlabs/
code/site \
    --hiera_config=/etc/puppetlabs/code/hiera.yaml \
    /etc/puppet/manifests/site.pp
```

We can mimic an ENC by placing, on our manifest file, all the top scope variables and classes that will be provided by it.

This usage pattern with Puppet is the most simple and direct and, curiously, is also a popular choice in some large installations; later, we will see how a Masterless approach, based on `puppet apply`, can be an alternative for scaling.

Default Puppet Master

A basic Puppet Master installation is rather straightforward: we just have to install the server package and we have what is needed to start working with Puppet:

- A `puppet master` service, which can start without any further configuration
- Automatic creation of the CA and of the Master certificates
- Default configurations that involve the following settings:
 - First manifest file processed by the Master in `/etc/puppetlabs/code/manifests/site.pp`
 - Modules searched in `/etc/puppet/modules` and `/opt/puppetlabs/puppet/modules`

Now, we just have to run Puppet on clients with `puppet agent -t --server <pupptmaster fqdn>` and sign their certificates on the Master (`puppet cert sign <client certname>`) to have a working client / server environment.

We can work with such a setup if we have no more than a few dozen nodes to manage.

We have already seen the elements that affect the Puppet Master's resources, but there is another key factor that should interest us: what are our acceptable catalog compilation and application times?

Compilation occurs on the Puppet Master and can last from few seconds to minutes; it is heavily affected by the number of resources and relationships to manage, but also, obviously, by the load on the Puppet Master, which is directly related to how frequently it has to compile catalogs.

If our compilation times are too long for us, we have to verify the following conditions:

- **Compilation time is always long** even with a single catalog processed at a time. In this case, we will have to work on two factors: **code optimization** and **CPU power**. Our manifests may be particularly resourceful and we have to work on how to optimize our code (we will see how later). We can also provide more CPU power to our Puppet Master as that is the most needed resource during compilation. Of course, we should verify its overall sanity: it shouldn't regularly swap memory pages to disk and have not faulty hardware that might affect performance. If we use stored configs, we should definitively use PuppetDB as the backend, either on the same server or in a dedicated one. We may also consider upgrading our Puppet version, especially if we are not using Puppet 4 yet.
- **Compilation time takes a long time because many concurrent catalogs are processed at the same time**. Our default Puppet Master setup can't handle the quantity of nodes that interrogate it. Many options are available in this case. We order them by ease of implementation:
 - **Reduce the frequency of each Puppet run** (the default 30 minutes interval may be longer, especially if we have a way to trigger Puppet runs in a centrally managed way, for example, via MCollective, so that we can easily force urgent runs).
 - If using a version older than Puppet 4, use **Apache Passenger** instead of the default web server.
 - **Have a multi Master setup** with load-balanced servers.

Puppet Master with Passenger

Passenger, known also as **mod_rails** or **mod_passenger**, is a fast application server that can work as a module with Apache or Nginx to serve Ruby, Python, Node.js, or Meteor web applications. Before version 4 and some of the latest 3.X versions, Puppet was a pure Ruby application that used HTTPS for client / server communication, and it could gain great benefits by using Passenger, instead of the default and embedded **Webrick**, as a web server.

The first element to consider when using older Puppet versions and there is the need to scale the Puppet Master is definitely the introduction of Passenger. It brings a pair of major benefits that are listed as follows:

- General **better performances** in serving HTTP requests (either via Apache or Nginx, which are definitively more efficient than Webrick)
- For **Multi CPU support**. On a standalone Puppet Master, there is just one process that handles all the connections, and that process uses only one CPU. With Passenger, you can have more concurrent processes that better use all the available CPUs.

On modern systems, where multiprocessors are the rule and not the exception, this leads to huge benefits.

Installing and configuring Passenger

Let's quickly see how to install and configure Passenger, using plain Puppet resources.

For the sake of brevity, we simulate an installation on a RedHat 6 derivative here. For other breeds, there are different methods to set up the source repo for packages, and possibly different names and paths for resources.

The following Puppet code can be placed on a file such as `setup.pp` and be run with `puppet apply setup.pp`.

First of all, we need to setup the EPEL repo, which contains extra packages for RedHat Linux that we need:

```
yumrepo { 'epel':
  mirrorlist => 'http://mirrors.fedoraproject.org/
mirrorlist?repo=epel-6&arch=$basearch',
  gpgcheck   => 1,
  enabled    => 1,
  gpgkey     => 'https://fedoraproject.org/static/0608B895.txt',
}
```

Then, we set up the Passenger's upstream yum repo of Stealthy Monkeys:

```
yumrepo { 'passenger':
  baseurl    => 'http://passenger.stealthymonkeys.com/
rhel/$releasever/$basearch',
  mirrorlist => 'http://passenger.stealthymonkeys.com/rhel/mirrors',
  enabled    => 1,
  gpgkey     => 'http://passenger.stealthymonkeys.com/RPM-GPG-KEY-
stealthymonkeys.asc',
}
```

We will then install all the required packages with the following code:

```
package { [ 'mod_passenger' , 'httpd' , 'mod_ssl' , 'rubygems' ] :  
  ensure => present,  
}
```

Since there is not a native RPM package, we install `rack`, a needed dependency, as a Ruby Gem:

```
package { 'rack':  
  ensure    => present,  
  provider => gem,  
}
```

We need also to configure an Apache virtual host file:

```
file { '/etc/httpd/conf.d/passenger.conf':  
  ensure => present,  
  content => template('puppet/apache/passenger.conf.erb')  
}
```

In our template (`$modulepath/puppet/templates/apache/passenger.conf.erb` would be its path for the previous sample), we need different things configured. The basic Passenger settings, which can eventually be placed in a dedicated file are as follows:

```
PassengerHighPerformance on  
PassengerMaxPoolSize 12 # Lower this if you have memory issues  
PassengerPoolIdleTime 1500  
PassengerStatThrottleRate 120  
RackAutoDetect On  
RailsAutoDetect Off
```

Then, we configure Apache to listen to the Puppet Master's port 8140 and create a Virtual Host on it:

```
Listen 8140  
<VirtualHost *:8140>
```

On the Virtual Host, we terminate the SSL connection. Apache must behave as a Puppet Master when clients connect to it, so we have to configure the paths of the Puppet Master's SSL certificates as follows:

```
SSLEngine on  
SSLProtocol all -SSLv2 -SSLv3  
SSLCipherSuite ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-  
POLY1305:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-  
SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:DHE-
```

```

RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-
SHA256:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-
AES256-SHA384:ECDHE-RSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA384:ECDHE-
ECDSA-AES256-SHA:ECDHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA256:DHE-
RSA-AES128-SHA:DHE-RSA-AES256-SHA256:DHE-RSA-AES256-SHA:ECDHE-ECDSA-
DES-CBC3-SHA:ECDHE-RSA-DES-CBC3-SHA:EDH-RSA-DES-CBC3-SHA:AES128-
GCM-SHA256:AES256-GCM-SHA384:AES128-SHA256:AES256-SHA256:AES128-
SHA:AES256-SHA:DES-CBC3-SHA:!DSS
    SSLCertificateFile /var/lib/puppet/ssl/certs/<%= @fqdn %>.pem
    SSLCertificateKeyFile /var/lib/puppet/ssl/private_keys/<% @fqdn
%>.pem
    SSLCertificateChainFile /var/lib/puppet/ssl/certs/ca.pem
    SSLCACertificateFile /var/lib/puppet/ssl/certs/ca.pem
    SSLCARevocationFile /var/lib/puppet/ssl/certs/ca_crl.prm
    SSLVerifyClient optional
    SSLVerifyDepth 1
    SSLOptions +StdEnvVars

```

 A good reference for recommended values for `SSLCipherSuite` can be found at <https://mozilla.github.io/server-side-tls/ssl-config-generator/>.

We also need to add some extra HTTP headers to the connection that is made to the Puppet Master in order to let it identify the original client (details on this later):

```

RequestHeader set X-SSL-Subject %{SSL_CLIENT_S_DN}e
RequestHeader set X-Client-DN %{SSL_CLIENT_S_DN}e
RequestHeader set X-Client-Verify %{SSL_CLIENT_VERIFY}e

```

Then, we enable Passenger and define a document root where we will create the rack environment to run Puppet:

```

PassengerEnabled On
DocumentRoot /etc/puppet/rack/public/
RackBaseURI /
<Directory /etc/puppet/rack/public/>
    Options None
    AllowOverride None
    Order allow,deny
    allow from all
</Directory>

```

Finally, we add normal logging directives as follows:

```
ErrorLog /var/log/httpd/passenger-error.log
CustomLog /var/log/httpd/passenger-access.log combined
</VirtualHost>
```

Then, we need to create the rack environment working directories and configuration as follows:

```
file { ['/etc/puppet/rack',
        '/etc/puppet/rack/public',
        '/etc/puppet/rack/tmp']:
  ensure => directory,
  owner  => 'puppet',
  group  => 'puppet',
}
file { '/etc/puppet/rack/config.ru':
  ensure => present,
  content => template('puppet/apache/config.ru.erb')
  owner  => 'puppet',
  group  => 'puppet',
}
```

In our config.ru, we need to instruct rack on how to run Puppet as follows:

```
# if puppet is not in your RUBYLIB:
# $LOAD_PATH.unshift('/opt/puppet/lib')
$0 = "master"
# ARGV << "--debug" # Uncomment to debug
ARGV << "--rack"
ARGV << "--confdir" << "/etc/puppet"
ARGV << "--vardir" << "/var/lib/puppet"
require 'puppet/util/command_line'
run Puppet::Util::CommandLine.new.execute
```

Once things are configured, we can start our Apache. However, before doing this, we need to disable the standalone Puppet Master service as it listens to the same 8140 port and would overlap with our Apache service:

```
service { 'puppetmaster':
  ensure => stopped,
  enable => false,
}
```

Then, we can finally start our Apache with Passenger. Remember that whenever we make changes to Puppet's configuration, the service to restart to apply them is Apache: Puppet Master standalone process should remain stopped:

```
service { 'httpd':  
  ensure => running,  
  enable => true,  
  require => Service['puppetmaster'], # We start apache after having  
    managed the puppetmaster service shutdown  
}
```

All this code, with the ERB templates it uses, should be placed in a module that allows autoloading of classes and files.

Puppet Master based on Trapperkeeper

One of the major changes in Puppet version 4 is that the Puppet Server is executed over a Java Virtual Machine. Ruby implementation was fine while it offered an agile development environment in the initial versions of the project, but as the software consolidates and Puppet language is more mature and stable, a better performance and improvements in scalability and speed were required.

Reimplementing a whole application to change the language doesn't seem to be a good idea, it is a huge effort that could block the evolution of the project. That's why, wisely, the Puppet Labs team decided to do it in a way that allowed them to reimplement just some of the parts at a time. JVM allows to execute the Ruby code with JRuby, so their first step was to make sure that Puppet worked with this interpreter while they implemented a services framework for JVM that could serve as a **glue** for the different parts implemented in different languages. This framework is known as **Trapperkeeper** and is implemented in **Clojure**.

This Trapperkeeper-based implementation offers by now two basic points of performance tuning: controlling the maximum number of JRuby instances running at a time, and controlling the memory usage of the whole application.

Puppet Server maintains a pool of JRuby instances. When it needs to execute some Ruby code, it picks up one of these instances till it is finished, and then releases it. If a request needs an instance and there are none available, then the request is blocked till one is released. So, if there are few instances, you can suffer a lot of contention in the requests to your Puppet Server, but also, with lots of them, the server can get overloaded. It's important to choose a good number of instances for your deployment.

The maximum number of instances can be controlled with the `max-active-instances` variable in the `puppetserver.conf` file. The default value of this setting (leaving it commented out in the configuration) makes the Puppet Server select a safe value based on the number of CPUs; but, depending on your deployment, you can see that the CPUs of your servers are underused, or that it's overloaded if it has more responsibilities. In that case, you can decide to evaluate some other values to see which one makes better use of your resources.

You also have to take into account the memory usage of the application. The more JRuby instances it has and the bigger your catalogs are, the more memory it will need. A recommendation is to assign 512 MB as a base and an additional 512 MB for each JRuby instance. If your Puppet catalogs are very big, or if your servers have spare memory to dedicate to Puppet Server, you may consider to increase the available memory. This setting has to be configured in the JVM start-up, with the parameters `-Xms` and `-Xmx` that respectively control the minimum and the maximum heap size. In a JVM most of the memory used is in the heap, but it will also need a little more memory, so leave some margin with the maximum available memory in the server. This value used to be configured in the defaults file (`/etc/sysconfig/puppetserver` or `/etc/defaults/puppetserver` depending on the distribution). For example, for a server with four instances in a server with 4 GB memory, and by applying the recommendation we could set it to 2560 MB, but it'd probably be safe to set it to 3 GB; a very adjusted value could trigger too many times the garbage collector, what penalizes CPU performance. This would be the setting for the defaults file:

```
JAVA_ARGS="-Xms3072m -Xmx3072m -XX:MaxPermSize=256m"
```

You can see that the `MaxPermSize` is also set; this limits the memory size of the permanent space, that is where the JVM stores classes, methods, and so on. Of course, any other settings available for the JVM could be used here.

Multi-Master scaling

A Puppet Server running on a decently sized server (ideal at least 4 CPUs and 4 GB of memory) should be able to cope with hundreds of nodes.

When this number starts to enter in to the range of thousands, or the compiled catalogs start to become big and complex, a single server will begin to have problems in handling all the traffic. Then, we need to scale horizontally, adding more Puppet Masters to manage clients' requests in a balanced way.

There are some issues to manage in such a scenario, the most important ones are:

- How to manage the CA and the server certificates
- How to manage SSL termination
- How to manage Puppet code and data

Managing certificates

Puppet's certificates are issued by a Certificate Authority, which is automatically created on the server when we start it the first time. We usually don't care much about it, we just sign certificate requests with `puppet cert` and have everything we need to work with clients.

On a multi-Master setup, an accurate management of the Puppet Certification Authority and of the Puppet Masters' certificates becomes essential.

The main element to consider is that the first time `puppet master` is executed, it automatically creates two different certificates, which are as follows:

- The CA certificate is used to sign all the other certificates:
 - The public key is stored in `/etc/puppetlabs/puppet/ssl/ca/ca_pub.pem`
 - The private key is in `/etc/puppetlabs/puppet/ssl/ca/ca_key.pem`
 - The certificate file is in `/etc/puppetlabs/puppet/ssl/ca/ca.crt.pem`
- The **Puppet Server's own host certificate** is used to communicate with clients:
 - The public key is stored in `/etc/puppetlabs/puppet/ssl/public_keys/<fqdn>`
 - The private key is stored in `/etc/puppetlabs/puppet/ssl/private_keys/<fqdn>`; the same paths are used on clients for their own certificates

On the Puppet Master, all the clients' public keys that still need to be signed by the CA are placed in `/etc/puppetlabs/puppet/ssl/ca/requests`, and the ones that have been signed are in `/etc/puppetlabs/puppet/ssl/ca/signed`.

The CA, which is managed via the `puppet ca` command, performs the following functions:

- Signs **Certificate Signing Requests (CSR)** from clients and transforms them in x509v3 certificates (when we issue `puppet cert sign <certname>`)
- Manages the **Certificate Revocation List (CRL)** of certificates we revoke with the `puppet cert revoke <certname>`
- Authenticates Puppet clients and Masters making them establish a trust relationship and communicate over SSL

There are a pair of important parameters that are related to certificates that should be considered in `puppet.conf` before launching the Puppet Master for the first time:

- `dns_alt_names`: This allows us to define a comma-separated list of names with which a node can be referred when using its certificate. By default, Puppet creates a certificate that automatically adds the names `puppet` and `puppet.$domain` to host's `fqdn`. We should be sure to have in this list of names both the local server hostname and the name the clients used to refer to the Puppet Master (probably associated with the IP of a load balancer).
- `ca_ttl`: This sets the duration, in seconds, of the certificates signed by the CA. The default value is 157680000, which means that after 5 years of starting your Puppet Master, its certificate expires and has to be reissued. This is an experience that most of us have already faced and involves the recreation and signing of all their certificates.



Note that the whole `/etc/puppetlabs/puppet/ssl` directory and the certificates it contains are recreated from scratch if it doesn't exist when Puppet runs. Therefore, if we want to recreate our Puppet Master's certificates with corrected settings, we have to move the existing `ssldir` to a backup place (just as a precaution in case we change the idea, we won't need it anymore otherwise), configure `puppet.conf` as needed, and restart the Puppet Master service.

This is an activity that we can do light heartedly, on the Master, only when it has been just installed and there are no (or few) signed clients because when we recreate `ssldir` with new certificates on the Master communication with clients won't be possible. All the previously signed certificates are no longer valid and have to be recreated.

CA management in a multi-master setup can be done in the following different ways:

- Configure one of the load balanced Puppet Masters as the CA server, and have all the other ones using it for CA activities. In this case, all the servers act as Puppet Servers and one of them also as the CA.
- Configure an external, eventually in High Availability, Puppet Master that provides only the CA service and is not used to compile clients' catalogs.

On `puppet.conf`, configuration is quite straightforward when the CA server is (or might be) different from the Puppet Master:

- On all the clients, explicitly set the `ca_server` hostname (which is, by default, the same Puppet Master):

```
[agent]
ca_server = puppetca.example42.com
```
- On the CA server, no particular configuration is needed:

```
[master]
certname = puppetca.example42.com
ca = true
```
- On the other Puppet Masters, we just have to define that the local server is not a CA and to look, as done for all the clients, for another `ca_server`:

```
[agent]
ca_server = puppetca.example42.com
[master]
certname = puppet01.example42.com
ca = false
```

Managing SSL termination

When we deal with Puppet's client server traffic, we can apply all the logics that are valid for HTTPS connections. We can, therefore, have different scenarios as follows:

- Clients' proxy (clients can use a proxy to reach remote or not directly accessible Puppet Masters)
- Master's reverse proxy (all clients communicate with frontend servers that proxy their requests to backend workers)
- Load balanced Masters at the network level (clients communicate directly with a load balanced server)
- Load balanced Master at application level (clients communicate with an intermediate host that balances and reverse proxies the Master)

When configuring the involved elements, we have to take care of the following elements:

- The SSL certificates used where the SSL connection is terminated must always be the ones of the Puppet Master and of the CA. If they are on different servers, we need to copy them.
- We have to communicate the client's name to the Puppet Master, and this is done by setting, where SSL is terminated, these HTTP headers: **X-SSL-Subject**, **X-Client-DN**, and **X-Client-Verify**, which indicate to the Master if the certificate is authenticated.

In our `puppet.conf` file, there are always the following default settings, which define the name of the HTTP header (with an `HTTP_` prefix and underscores instead of dashes).

They contain the clients' SSL **Distinguished Name (DN)** and the name of the HTTP header that contains the status message of the client verification (expected value for a trusted, not revoked, client certificate is `SUCCESS`):

```
ssl_client_header = HTTP_X_CLIENT_DN
ssl_client_verify_header = HTTP_X_CLIENT_VERIFY
```

On the web server(s) where SSL is terminated (it might be Passenger in a single server setup or an Apache, which balances and reverse proxies the Puppet Master backend), we need to set these HTTP headers extracting info from SSL environment variables as follows:

```
RequestHeader set X-SSL-Subject %{SSL_CLIENT_S_DN}e
RequestHeader set X-Client-DN %{SSL_CLIENT_S_DN}e
RequestHeader set X-Client-Verify %{SSL_CLIENT_VERIFY}e
```

These servers are the ones that communicate directly with clients and terminate the SSL connection, we can define them as **frontend** servers, they act as proxy and generate a new connection to **backend** Puppet Masters that do the real work and compile catalogs.

Since SSL has been terminated on the frontends, traffic from them to backend servers is in clear text (they are supposed to be in the same LAN), and on the backend Apache, we need to state where to get the client's certificates DN, using the previous extra headers:

```
SetEnvIf X-Client-Verify "(.*)" SSL_CLIENT_VERIFY=$1
SetEnvIf X-SSL-Client-DN "(.*)" SSL_CLIENT_S_DN=$1
```

Also, on a backend server, we do not need to configure all the other SSL settings, and just need a Virtual Host with rack configurations.

Given this information, we can compose our topology of web servers that handle Puppet traffic in a very flexible way, with one or more frontend servers that proxy requests to the backend Puppet Masters and terminate SSL, and with backend Puppet Masters that run Puppet via Passenger.

Managing code and data

Deployment of Puppet code and data is another factor to consider. We probably want the same code deployed on all our Puppet Masters. We can do this in various ways: all of them basically require the remote and/or triggered execution of some commands (if we want to avoid the need to log into each server every time a change on Puppet is done) or a way to keep files synced across different servers.

How a deployment script or command may work is definitively tied to how we manage our code: we might execute `r10k` or `librarian puppet`, or make a `git pull` on our local directories to fetch changes from a central repo.

Alternatively, we might decide to have our Puppet code and data on a shared file system or keep them synced with tools such as `rsync`.

In any case, we have to copy/sync or share all the directories where our code and data: the **modules**, **manifest**, and **Hiera** directories, if used, are placed.

Load balancing alternatives

When we have to balance a pool of Puppet Masters, we have different options, which are as follows:

- HTTP load balancing with SSL termination is done on the load balancer, which then proxies clients' requests to the Puppet Masters.
- TCP load balancing with SSL termination is done on the Puppet Masters that directly communicate with clients. In this case, the load balancer (which may be a software like `haproxy` or a dedicated network equipment) listens to the Virtual IP used by the clients to contact the Master (for example, a common `puppet.example42.com`). It then redirects all the TCP connections to the Puppet Masters (in their `dns_alt_names` they need to have the name of the Puppet Master host configured on clients).

- DNS round robin can be considered the poor man's alternative to TCP load balancing. Clients are configured to use a single hostname for the Puppet Master, which is resolved, via DNS, to the multiple Masters' addresses. Also in this case, SSL connections are terminated directly on the Masters and they must have the name used by clients in their `dns_alt_names`. This solution is quite easy to implement, as it does not require additional systems to manage load balancing, but has the (major) drawback of not being able to detect failures and remove non-responding Puppet Masters from the pool of balanced servers.
- **DNS SRV** records can also be used to define the IP addresses of the Puppet Masters via DNS, allowing the possibility to set priorities and fail overs. This feature is available only on Puppet 3 and later. To use this option, instead of using the `server` parameter in `puppet.conf`, we have to indicate the `srv_domain` this way:

```
[main]
  use_srv_records = true
  srv_domain = example.com
```



DNS SRV records are used to define the hostnames and ports of servers that provide specific services. They can also set priorities and weights for the different servers. For example, for Puppet, the following records could be used:

```
_x-puppet._tcp.example.com. IN SRV 0 5 8140
p1.example.com.
_x-puppet._tcp.example.com. IN SRV 0 5 8140
p2.example.com.
```

Clients need to explicitly support these records in order to use these kind of configurations.

Masterless Puppet

An alternative approach to the Puppet Master scaling methods that we have seen so far is not to use it at all. Masterless setups involve the direct execution of `puppet` apply on each node, where all the needed Puppet code and data has to be stored.

In this case, we have to find a way to distribute our modules, manifests, and, eventually, Hiera data to all the clients. We still can use external components such as:

- **ENC**: The `external_nodes` script can work as it works on the Puppet Master; it can interrogate any external source of knowledge on how to classify nodes. A concern here is whether it makes sense to introduce a central point of authority when we want a distributed decentralized setup.

- **Report:** Also, the reporting function can work exactly as it works on Puppet Master. Here, as for the ENC, the basic difference is that whatever the tool used, it must allow access from any node in our infrastructure, and not just the Master.
- **Exported Resources:** This can be used too, with some caveats. If we use the active records' backend, we need to access the database from all the nodes. If we use PuppetDB, we need to establish a trust between the certificates of the PuppetDB server and the ones of each client.

We also need a way to run Puppet on the clients in an automated or centrally managed way; it may be via a cron job or a remote command execution.

Distribution of Puppet code and data may be done in different ways, as follows:

- Executing `git pull` from central repositories
- Update of native packages (`rpm`, `deb` and so on) from a custom repo
- Running a command such as `sync` or `rdiff`
- Mount from NFS or another network or shared filesystem
- Bit Torrent with tools such as Murder (<https://github.com/lg/murder>)

Configurations and infrastructure optimizations

Whatever the layout of our Puppet infrastructure, we may consider some other options to optimize its performances.

Traffic compression

A first quick attempt may be done by activating the compression of HTTPS traffic between clients and Master. The following option has to be set on `puppet.conf` at both ends:

```
http_compression = true
```

The case where it makes sense to enable it is mostly where we have clients that reach the server via a WAN link, generally via a VPN, where throughput is definitely not the one we have on LAN communications. If we have large catalogs and reports, their compression during transfer, being mostly text files, can be quite effective.

Caching

Another area where we might operate is catalog caching. This is a delicate topic, as it is not easy to determine what has changed on the client's side (some facts like `uptime` change always by definition, others are supposed to be more stable) and on the server's side (changes on the Puppet code and data may or may not affect a specific node). The challenge therefore, is to always provide the correct and updated catalog when a caching mechanism is in place.

Puppet provides some configuration options to manage caching. By default, Puppet doesn't recompile the catalog if it has a local version cached with an updated timestamp and facts which have not changed. When we want to be sure to obtain a new catalog, we have to enable the `ignorecache` option:

```
ignorecache = true # Default: false
```

Note that this is automatically done when we run the `puppet agent -t` command, which ensures that we have always a freshly compiled catalog.

We can also tell to the client to always use a local cached copy of the catalog, instead of asking it to the Puppet Master:

```
use_cached_catalog = true # Default: false
```

This might be useful in cases where we want to temporarily **freeze** the configurations applied to a client without having to disable the Puppet service and without caring about eventual changes on the Puppet Master.

Distributing Puppet run times

If we run Puppet via cron or other time based mechanism, we need to avoid the problem of having all our clients hitting the Master and requesting their catalog at the same time. There are various options to distribute Puppet runs in order to avoid peaks of too many concurrent requests.

We can introduce a random sleep delay in the command we execute via cron, for example with cron entries based on ERB templates, such as:

```
0,30 * * * * root sleep <%= @sleep %> ; puppet agent --onetime
```

Where the `$sleep` variable with the number of seconds to wait may be randomly defined in Puppet manifests with the `fqdn_rand()` function, which returns a random value based on the node's full hostname (so it's random (not in a cryptographically usable way), but doesn't change at every catalog compilation):

```
$sleep = fqdn_rand('1800') # Returns a number from 0 to 1800
```

Alternatively, we can use the `splay` configuration option in `puppet.conf`, which introduces a random (but consistent) delay at every Puppet run, and which can be as long as defined by `splaylimit` (whose sane default is Puppet's run interval):

```
splay = true # Default: false
splaylimit = 1h # Default: $runinterval
```

File system changes check

On Puppet Master, there is an option, `filetimeout`, which sets the minimum time to wait between checking for updates in configuration files (manifests, templates, and so on). This determines how quickly the Master checks whether a file is changed on disk.

The default value is 15 seconds, and can be changed in `puppet.conf`.

This setting has very limited effects on the performances (unless, I suppose, we lower it too much), but it's important to know that it exists, because it's the reason why, sometimes, nothing new happens on the client when we launch a Puppet run immediately after a change on some file of Puppet Master.

This may lead to some confusion, we make a change on some manifests, we run Puppet, and nothing happens. Then, we run Puppet again and the change is finally received and we wonder what the hell is happening. Therefore, be aware that there is such an option and, more importantly, be aware of this behavior of the Master that **scans** the directories where our Puppet code and files are placed at regular intervals and might not immediately process the very latest changes made to these files.

Scaling stored configs

We have seen that the usage of exported resources allows resources declared on a node to be applied on another node. In order to achieve this, Puppet needs the `storeconfigs` option enabled and this involves the usage of an external database where all the information about the exported resources is stored.

The usage of stored configs has been historically a big performance killer for Puppet. The amount of database transactions involved for each run makes it a quite resource intensive activity.

There are various options in `puppet.conf` that permit us to tune our configurations. The default settings are as follows:

```
storeconfigs = false
storeconfigs_backend = active_record
dbadapter = sqlite3
thin_storeconfigs = false
```


If we enable them with `storeconfigs = true`, the default configuration involves the usage of the `active_record` backend and a SQLite database.

This is a solution that performs quite badly and therefore should be used only in test or small environments. It has the unique benefit that we don't need any other activity, we just have to install the SQLite Ruby bindings package on our system. With such a setup, we will quickly have access problems to the SQL backend with multiple concurrent Puppet runs.

The next step is to use a more performant backend for data persistence. Before the introduction of PuppetDB, MySQL was the only alternative. In order to enable it, we have to set the following options in `puppet.conf`:

```
dbadapter = mysql
dbname = puppet          # Default value
dbserver = localhost     # Default value
dbuser = puppet          # Default value
dbpassword = puppet      # Default value
```

Such a setup involves a local MySQL server where we have created a puppet database with the relevant grants, so from our MySQL console, we should write something like the following code:

```
create database puppet;
GRANT ALL ON puppet.* to 'puppet'@'localhost' IDENTIFIED by 'puppet';
flush privileges;
```

This is enough to have a Puppet Master storing its data on a local MySQL backend. If the load on our system increases, we can move the MySQL service to another dedicated server and can tune our MySQL server.

Brice Figureau, who heavily contributed to the original store configs code, made an interesting presentation at the first Puppet Camp on this topic at <http://www.slideshare.net/masterzen/all-about-storeconfigs-2123814>, where useful hints are provided to configure MySQL in a dedicated server to scale for the inserts:

```
innodb_buffer_pool_size = 70% of physical RAM
innodb_log_file_size = up to 5% of physical RAM
innodb_flush_method = O_DIRECT
innodb_flush_log_at_trx_commit = 2
```

Also, to optimize the most common queries on Puppet's Wiki, it is suggested that this index is created from the MySQL console as follows:

```
use database puppet;
create index exported_restype_title on resources (exported, restype,
title(50));
```

We can limit the amount of information stored by setting `thin_storeconfigs = true`. This makes Puppet store just facts and exported resources on the database and not the whole catalog and its related data. This option is useful with the `active_record` backend (with PuppetDB it is not necessary).

What we have written so far about store configs using the active records backend made a lot of sense some years ago, and we referenced it here to have a view on how to scale with store configs. Truth is that the best and recommended way to use store configs is via the PuppetDB backend, this is done by placing these settings in `puppet.conf`:

```
storeconfigs = true
storeconfigs_backend = puppetdb
```

We have dedicated the whole of *Chapter 3, Introducing PuppetDB* to PuppetDB because it is definitively a major player in the Puppet ecosystem. The performance improvements it brings are huge so there is really no reason not to use it.

The components of PuppetDB can be distributed to scale better:

- PuppetDB can be horizontally scaled. It's a stateless service entirely based on a REST like interface. Different PuppetDB servers can be load balanced either at TCP or HTTP level.
- PostgreSQL server may be moved to a dedicated host and then scaled or configured in the high availability mode following PostgreSQL's best practices.

Measuring performance

When we start to have a remarkable number of resources on a node (in the order of several hundreds or thousands), compilation and application time of a node catalog grows to uncomfortable levels.

If the number of the nodes to manage is big, even small tunings and optimizations of our code can bring interesting results.

For this reason, it is useful to have tools and techniques that permit us to measure Puppet's performance metrics at our disposal.

Puppet Metrics

Puppet itself provides some options that help us understand where time is spent during its activities.

At the end of each Puppet run, it is possible to see a detailed report on the time spent for each kind of activity; on `puppet.conf`, we can enable reports with the following option:

```
report = true # Enable client's reporting
```

We can have a summary of the run times with the following option:

```
summarize = true # Print a summary of the Puppet transaction
```

At the end of a Puppet run, we can have metrics that let us understand how much time the server spent in compiling and delivering the catalog (**Config retrieval time**), how much has been spent to manage each of the most common types of resources: package installation, files management, commands executions, and so on.

Of these metrics, the one related to config retrieval time is probably the most interesting as it is directly related to the work that the Puppet Master has to do.

This key metric can be retrieved directly on the Puppet Master with a quick glance to the logs, where the compilation time for each generated catalog is reported. On **RedHat** based systems, we can get this with:

```
grep 'Compiled catalog' /var/log/messages
```

On other distros or OSes, just look for the log file where syslog stores messages with the daemon facility (or what's configured by the option `syslogfacility`).

If we want to see how much time Puppet takes to evaluate each resource of the catalog, we can enable the option:

```
evaltrace = true
```

This is actually more for performance reasons (evaluation times should always be in the order of 0.0x seconds) and might be useful when we want to see Puppet's exact order in evaluating and applying resources.

Since Puppet 3.4.0, we also have at our disposal the very useful `--profile` option, which gives a lot of useful information for troubleshooting performance issues. Please refer to <http://gatling-tool.org/> and <http://puppetlabs.com/blog/puppet-gatling-and-jenkins-together> for more details.

Optimizing code

Every Puppet user complains about catalog compilation times. That's a fact. Sooner or later, seldom or always, depending on our patience and time, we have our moment of frustration for how much time it takes to churn out the catalogs of our nodes. There is something we can do for this.

The first basic rule is that the more resources we manage, the longer the Puppet is run: for each resource there's something to parse and compile on the Master, write in the catalog, send to clients, apply locally, report back to the Master, and handle to the report backend.

This is hardly an issue for few resources, but when we have nodes with several hundred, or sometimes thousands of them; things definitely change.

The overall number of resources managed in our nodes can grow with these factors:

- Extension of Puppet coverage on more services and managed resources. This is quite obvious: the more components of a system we have to manage with Puppet, the more relevant resources we will have to declare in our manifests.
- A single defined type used many, too many, times. For example, if we manage local system users via Puppet and we have many of them, or we have a server with hundreds or thousands of Apache Virtual Hosts managed by a specific define.
- If we decide to manage configuration files with a setting-based approach (for example, with Augeas or other in-file line management defines) the number of resources of our nodes can explode easily (one resource for each line of each configuration file of each node; the result of these multiplying factors can easily get out of control).
- An excessive use of classes and subclasses that fragment our code. For example, in my very personal opinion, the pattern that suggests the division of a module in three major subclasses (for example, `openssh::package`, `openssh::service`, and `openssh::config`) makes little sense when we have only one resource for each subclass. A small module that manages a typical `package-service-config` application can have its overall number of resource raised from 4 (the main class and the package, service, and file resources) to 7 (all of these plus the three containing subclasses).

When we have the same resource type used many times on the same catalog, we may study alternatives that may deliver great benefits for our performances:

- When they are used to manage setting-based configuration files (`augeas` or `file_line` from the `stdlib` module) or others, we may question whether it makes sense to manage those files in this way or use a simple ERB template, eventually using there parameters such as `config_file_hash` that allows us to manage any custom configuration entry in a file via a hash.
- When a single `define` is used to create fragments of configuration files, such a hypothetical `apache::vhost`, we might evaluate the usage of a function that returns the whole content of a single huge file based on a data source with the information about the whole virtual host. The result would be a single resource instead of many.

Besides optimizations on the number of resources, we can consider few other general recommendations as follows:

- Do not use the `file` type to deliver very large files or binaries: For each file, Puppet has to make a checksum to verify that it has changed. For such cases we use `packages` instead.
- Whenever the `source =>` argument is used to provide a file, a new connection is made to Puppet Master during catalog application time, with `content =>` instead, the whole content of the file is placed in the catalog.
- Avoid too many elements in a source array for file retrieval, as follows:

```
source => [ "site/openssh/sshd.conf---$::hostname" ,  
           "site/openssh/sshd.conf--$::environment-$::role" ,  
           "site/openssh/sshd.conf-$::role" ,  
           "site/openssh/sshd.conf" ],
```

This checks three files (and eventually gets 3, 404 errors from the server) before getting the default ones.



When we have many files provided by the Puppet Master's fileserver (whenever we use the `source` argument in a file type), we might evaluate the opportunity of moving the fileserver functionality to a separate dedicated node. Here, we can setup a normal Puppet Master, which just serves static files and is not used to compile catalogs. We can then refer to it in our code with something like:

```
file { ['/tmp/sample':
  source => "puppet://$filesERVER_name/path/to/file",
}
```

Testing different Puppet versions

In the `ext/` directory of the Puppet code repository, there is `envpuppet`, a smart bash script written by Jeff McCune that makes it easy to test different Puppet versions.

Usage is easy; we create a directory and clone the official Puppet repos from GitHub using the following commands:

```
cd /usr/src
git clone git://github.com/puppetlabs/puppet.git
git clone git://github.com/puppetlabs/facter.git
git clone git://github.com/puppetlabs/hiera.git
```

Then, we can switch to the version we want to test using the following command:

```
cd /usr/src/puppet && git checkout tags/3.0.2
cd /usr/src/facter && git checkout tags/1.6.17
```

We then set an environment variable that defines the `basedir` directory for `envpuppet` using the following command:

```
export ENVPUPPET_BASEDIR=/usr/src
```

Now, we can test Puppet prepending `envpuppet` to any Puppet, Facter, or Hieria command:

```
envpuppet puppet --version
envpuppet facter --version
```

Alternatively, it can be possible to use the `code configuration` parameter to use `envpuppet` as default executable for Puppet.

Summary

In this chapter, we have seen how Puppet can scale while our infrastructure grows. We have to consider all the components involved.

For testing or small environments, we may have an all-in-one server, but it makes sense to separate these components from the beginning on dedicated nodes for the Puppet Server, for PuppetDB and its backend database (we might decide to move the PostgreSQL service to a dedicated server too), and eventually for an ENC.

When we need to scale further, or want high availability on these components, we can start to scale out horizontally and load balance the Puppet Master and PuppetDB systems (they all provide stateless HTTP(s) services) and cluster our database services (following the available solutions for PostgreSQL and MySQL).

When the bottleneck of a centralized Puppet Server becomes an issue or simply a not preferred solution, we might decide to go Masterless, so we have all our clients compiling and running their own manifests independently, without overloading a central server. But it might be more complex to set up, and may add some security concerns (Puppet data about all nodes is stored on all the other nodes).

Besides changes at the infrastructure level, we can scale better if our code performs well.

We have seen how to measure Puppet times and where code and configuration tuning may improve performances. In particular, we have reviewed the first, obvious, and basic element that affects Puppet times – the number of managed resources for a node. We also reviewed how we can deal with edge cases where some of them are used multiple times.

In the next chapter, we are going to dive more deeply into Puppet internals and discover how we can extend its functionalities.

10

Extending Puppet

Puppet is impressively extendable. Almost every key component of the software can be extended and replaced by code provided by users.

Most of the time we can use modules to distribute our code to clients, but Puppet goes further; surprising things are possible with the **indirector** and its **termini** (somehow strange words that are going to be clearer in the next pages).

This chapter is about understanding and extending Puppet code. We are going to review the following topics:

- Anatomy of a Puppet run, what happens under the hood
- What are Puppet extension alternatives?
- Developing custom functions
- Developing custom facts
- Developing custom types and providers
- Developing custom reports handlers
- Developing custom faces

The subject is large, we are going to have an overview and show examples that let us dive in some details. For more information about how Puppet works and its inner beauties, check these great sources, in detail:

- The Puppet Extension Points and Puppet Internals series of blog posts by Brice Figuerau. They are by far the most intriguing, fascinating, and better explained sources of documentation about Puppet's inner workings (<http://www.masterzen.fr>).
- Puppet on the edge, the blog of Puppet Labs' Henrik Lindberg, focused on the new features of Puppet that we will see in *Chapter 12, Future Puppet* <http://puppet-on-the-edge.blogspot.it/>.

- The puppet-dev discussion group, where ideas are discussed and questions are answered can be found at <https://groups.google.com/forum/#!forum/puppet-dev>.
- The official developer reference is at <https://docs.puppetlabs.com/references/latest/developer/>.
- The official ticketing system, where we can see how the actual discussion and work on the code is delivered: <https://tickets.puppetlabs.com>.

Anatomy of a Puppet run, under the hood

We have seen the output of a Puppet run in *Chapter 1, Puppet Essentials*, now let's explore what happens behind those messages.

We can identify four main stages that turn our Puppet code in a catalog that is applied on clients:

- **Parsing and compiling:** This is where Puppet manifests are fed to the `Puppet::Parser` class, which does basic syntax checks and produce an **Abstract Syntax Tree (AST)** object. This represents the same objects we defined in our manifests in a machine-friendly format. Both the facts, received from the client, and the AST are passed to the compiler. Facts and manifests are interpreted and the result converted into a tree of transportable objects: the resource catalog, commonly called **catalog**. This phase happens on the server unless we use the `puppet apply` command.
- **Transport:** In this phase, the Master serializes the catalog in the PSON format (a Puppet version of JSON), and sends it over HTTPS to the client. Here, it is deserialized and stored locally. The transport phase doesn't occur in a Masterless setup.
- **Instantiation:** Here, the objects present in the resource catalog (instances of `Puppet::Resource`) are converted in instances of the `Puppet::Type` class and a **Resource Abstraction Layer (RAL)** catalog is generated. Note that resource catalog and RAL catalog are two different things.
- **Configuration:** This is where the real action happens inside a Puppet transaction. The RAL catalog is passed to a new `Puppet::Transaction` instance; relationships and resources are evaluated and applied to the system.
- **Report.** A report of the transaction is generated on the client and eventually sent back to the Master.

The involved nodes (client / Master), components, actions, and classes are summarized in the following table (these steps refer generally to Puppet 3 and 4, may change on versions):

Node	Component	Action	Class#method
Client	Configurer	Plugins are downloaded and loaded	Puppet::Configurer::PluginHandler#download_plugins
Client	Configurer	Local facts are collected and sent to the Master	Puppet::Configurer#prepare_and_retrieve_catalog
Master	Compiler	Compilation is started by the indirection of the REST call	Puppet::Resource::Catalog::Compiler
Master	Parser	Manifests are parsed and an AST is generated	Puppet::Parser::Parser#parse
Master	Compiler	From this AST a graph of Puppet resources is elaborated	Puppet::Parser::Compiler#compile
Master	Compiler	The output of this operation is the resource catalog	Puppet::Resource::Catalog
Master	Network	Catalog is serialized as a PSON object and sent over the network	Puppet::Network::HTTP::API::V1#do_find Puppet::Network::HTTP::API::IndirectedRoutes#do_find
Client	Configurer	Catalog is received, deserialized, and cached locally	Puppet::Configurer#prepare_and_retrieve_catalog
Client	Configurer	Catalog is transformed to a RAL catalog	Puppet::Type
Client	Transaction	Each instance of Puppet::Type in the catalog is applied	Puppet::Transaction#evaluate

Node	Component	Action	Class#method
Agent	Configurer	Transaction report is saved to the configured report termini	Puppet::Configurer#send_report

Puppet extension alternatives

Extendibility has always been a widely pursued concept in Puppet; we can practically provide custom code to extend any activity or component of the software.

We can customize and extend Puppet's functionalities in many different ways operating at different levels:

- Key activities such as nodes classification via an ENC or variables definition and management via Hiera can be customized to adapt to most of the users' needs.
- Our code can be distributed via modules, using the `pluginsync` functionality. This is typically used to provide our facts, types, providers, and functions but it basically may apply to any piece of Puppet code.
- We can configure indirections for the main Puppet subsystems, and use different backends (called `termini`) to manage where to retrieve their data.

ENC and Hiera Extendibility

We have already seen, in earlier chapters of this book, how it is possible to manage in many different ways some key aspects of our Puppet infrastructure:

- We can manage the location where our nodes and the classes are defined and which variables and environments they use. This can be done via an ENC, which may be any kind of system that feeds us this data in the YAML format via the execution of a custom script, which may interrogate whatever backend we may have.
- We can manage the location where our Hiera data is stored, having the possibility to choose from many different backends for data persistence.

We are not going to talk again about these extension possibilities, we just have to remember how powerful they are and how much freedom they give us in managing two key aspects of our Puppet infrastructures.

Modules pluginsync

When in our `puppet.conf`, we set `pluginsync=true` (this is default from Puppet 3, but on earlier versions it has to be explicitly set), we activate the automatic synchronization of plugins. The clients, when Puppet is invoked and before doing any other operation, retrieve the content of the `lib` directories on all modules from the Master in their `modulepath` and copy them in their own `lib` directory (`/var/lib/puppet/lib` by default), keeping the same directory tree.

In this way, all the extra plugins provided by modules can be used on the client exactly as core code. The structure of the `lib` directory of a module is as follows:

```
{modulepath}
└─ {module}
   └─ lib
      ├── augeas
      │   └─ lenses
      ├── hiera
      │   └─ backend
      ├── puppetdb
      ├── facter
      └─ puppet
         ├── parser
         │   └─ functions
         ├── provider
         │   └─ $type
         ├── type
         ├── face
         └─ application
```

The preceding layout suggests that we can use modules to distribute custom facts to clients – Augeas lenses, faces, types and providers. We can also distribute custom functions and Hierarchical backends, but this is not useful on clients since they are used during the catalog compilation phase, which usually occurs on the Master (the notable exception is in Masterless setups when `pluginsync` is not necessary).

A small intriguing note is that the plugins synchronization done at the beginning of a Puppet run is actually performed using a normal file type, which looks like this:

```
file { $libdir:
  ensure => directory,
  recurse => true,
  source  => 'puppet:///plugins',
  force   => true,
  purge   => true,
  backup  => false,
```

```
noop      => false,  
owner     => puppet, # (The Puppet process uid)  
group     => puppet, # (The Puppet process gid)  
}
```

Similarly, all the files and directories configured in `puppet.conf` are managed using normal Puppet resources, which are included in a small settings catalog that is applied on the client as the very first step.

Some notes about the preceding file resource are as follows:

- The `purge` and `recurse` arguments ensure that removed plugins on the master are also removed on the client, and new ones are added recursively
- The `noop => false` parameter ensures that a regular `pluginsync` is done, even when we want to test a Puppet run with the `--noop` argument passed at the command line.
- The `source => puppet:///plugins` source is based on an automatic filesystem mount point that maps to the `lib` directory of every module. This can be modified via the configuration entry `pluginsource`.
- The `$libdir` can be configured via the homonymous configuration entry.

In this chapter, we are going to review how to write the most common custom plugins, which can be distributed via user modules: functions, facts, types, providers, report handlers, and faces.

Puppet indirector and its termini

Puppet has different subsystems to manage objects such as catalogs, nodes, facts, and certificates. Each subsystem is able to retrieve and manipulate the managed object data with REST verbs such as `find`, `search`, `head`, and `destroy`.

Each subsystem has an **indirector** that allows the usage of different backends, called **termini**, where data is stored.

Each time we deal with objects such as nodes, certificates, or others, we can see in the table later we work on an instance of a model class for that object, which is indirected to a specific **terminus**.

A terminus is a backend that allows retrieval and manipulation of simple key-values related to the indirected class.

This allows the Puppet programmer to deal with model instances without worrying about the details of where data comes from.

Here is a complete list of the available indirections, the class they indirect, and the relevant termini:

Indirection	Indirected Class	Available termini
catalog	Puppet::Resource::Catalog	active_record, compiler, json, queue, rest, static_compiler, store_configs, and yaml
certificate	Puppet::SSL::Certificate	ca, disabled_ca, file, and rest
certificate_request	Puppet::SSL::CertificateRequest	ca, disabled_ca, file, memory, and rest
certificate_revocation_list	Puppet::SSL::CertificateRevocationList	ca, disabled_ca, file, and rest
certificate_status	Puppet::SSL::Host	file and rest
data_binding	Puppet::DataBinding	hiera and none
facts	Puppet::Node::Facts	active_record, couch, facter, inventory_active_record, inventory_service, memory, network_device, rest, store_configs, and yaml
file_bucket_file	Puppet::FileBucket::File	file, rest, and selector
file_content	Puppet::FileServing::Content	file, file_server, rest, and selector
file_metadata	Puppet::FileServing::Metadata	file, file_server, rest, and selector
key	Puppet::SSL::Key	ca, disabled_ca, file, and memory

Indirection	Indirected Class	Available termini
node	Puppet::Node	active_record, exec, ldap, memory, plain, rest, store_configs, yaml, and write_only_yaml
report	Puppet::Transaction::Report	processor, rest, and yaml
resource	Puppet::Resource	active_record, ral, rest, and store_configs
resource_type	Puppet::Resource::Type	parser and rest
status	Puppet::Status	local and rest



For the complete reference, check out: <http://docs.puppetlabs.com/references/latest/indirection.html>

Puppet uses the preceding indirections and some of their termini every time we run it; for example, in a scenario with a Puppet Master and a PuppetDB these are the indirections and termini involved:

- On the agent, Puppet collects local facts via **facter**:
facts find from terminus facter
- On the agent, a catalog is requested from the Master:
catalog find from terminus rest
- On the master, facts are saved to PuppetDB:
facts save to terminus puppetdb
- On the master, node classification is requested from an ENC:
node find from terminus exec
- On the master, catalog is compiled:
catalog find from the terminus compiler
- On the master, catalog is saved to PuppetDB:
catalog save to terminus puppetdb
- On the agent, the received catalog is applied and a report is sent:
report save to terminus rest
- On the master, the report is managed with the configured handler:
report save to terminus processor

We can configure the termini to use each indirector either with specific entries in `puppet.conf` or via the `/etc/puppetlabs/puppet/routes.yaml` file, which overrides any configuration setting.

On `puppet.conf`, we have these settings by default:

```
catalog_cache_terminus =
catalog_terminus = compiler
data_binding_terminus = hiera
default_file_terminus = rest
facts_terminus = facter
node_cache_terminus =
node_terminus = plain
```

Just by looking at these values, we can deduce interesting things:

- The ENC functionality is enabled by specifying `node_terminus = exec`, which is an alternative terminus to fetch the resources and parameters to assign to a node.
- The catalog is retrieved using the Puppet `compiler` by default, but we might use other termini, such as `json`, `yaml`, and `rest` to retrieve catalogs from local files or remote REST services. This is exactly what happens with the `rest` terminus, when an agent requests a catalog to the Master.
- Hiera lookups for every class parameter (the **data binding** functionality introduced on Puppet 3) that can be replaced with a custom terminus with other lookup alternatives.
- The `*_cache_terminus` are used, for some cases, as secondary termini in case of failures of the default primary ones. The data retrieved from the default terminus is always written to the correspondent cache terminus in order to keep it updated.

In *Chapter 3, Introducing PuppetDB*, we have seen how a modification to the `routes.yaml` file is necessary to setup PuppetDB as the backend for the facts terminus; let's use it as an example:

```
---
master:
  facts:
    terminus: puppetdb
    cache: yaml
```

Note that we refer to the Puppet mode (`master`), the indirection (`facts`), and terminus and their eventual cache. Other termini that do not have a dedicated configuration entry in `puppet.conf` may be set here as needed.

We can deal with indirectors and their termini when we use the puppet command; many of its subcommands refer directly to the indirectors reported in the previous table and let us specify, with the `--terminus` option, which terminus to use.

We will come back on this later in this chapter when we will talk about Puppet faces.

Custom functions

Functions are an important area where we can extend Puppet. They are used when Puppet parses our manifests and can greatly enhance our ability to fetch data from custom sources, filter, and manipulate it.

We can distribute a function just by placing a file at `lib/puppet/parser/<function_name>.rb` in a module of ours.

Even if they are automatically distributed to all our clients, it's important to remember that being used only during the catalog compilation, functions are needed only on the Puppet Master.



Note that since they are loaded in the memory when Puppet starts, if we change a function on the Master, we have to restart its service in order to load the latest version.

There are two kinds of functions:

- `:rvalue` functions return a value, they are typically assigned to a variable or a resource argument. Sample core `rvalue` functions are `template`, `hieradata`, `regsubst`, `versioncmp`, and `inline_template`.
- `:statement` functions perform an action without returning any value. Samples from Puppet core are `include`, `fail`, `hieradata_include`, and `realize`.

Let's see how a real function can be written, starting from a function called `options_lookup` that returns the value of a given key of a hash.

Such a function is quite useful in ERB templates where we want to use arbitrary data provided by users via a class parameter as a hash.

We can place it inside any of our modules in a file called `lib/puppet/parser/options_lookup.rb`.

The first is the `newfunction` method of the `Puppet::Parser::Functions` class. Here, we define the name of the function and its type (if it's `rvalue` we have to specify it, by default a function's type is `statement`):

```
module Puppet::Parser::Functions newfunction(:options_lookup, :type =>
  :rvalue, :doc => <<-EOF
```

A description of what the function does and how it can be used is passed with the `doc` argument; here, the whole description that is inside a `doc` block is ended by `EOF`.

This function takes two arguments (`option`, and the default value) and looks for the given option key in the calling module's option's hash, and returns the value. [...]:

```
EOF
  ) do |args|
```

After the parameters are passed to the `newfunction` method, we have the function body, here we catch all the arguments passed to our function in the `args` variable.

We can raise an error if their number is not what's expected (2 or 3):

```
    raise ArgumentError, ("options_lookup(): wrong number of arguments
  ({args.length}; must be 2 or 3)") if (args.length != 2 and args.
  length != 3)
```

Then, we assign local variables to each argument, note that, since `args` is managed as an array, the first argument passed to our function is referred by `args[0]`, we also assign to our `mod_name` variable the value of Puppet's `parent_module_name` internal variable:

```
    value = ''
    option_name = args[0]
    default_val = args[1]
    hash_name = args[2]
    mod_name = parent_module_name
```

We set the default name (`options`) of the class parameter that contains the hash to use, note that we have had to cope with different Puppet versions where a missing variable is referenced in different ways (with the `:undefined` symbol, an empty string, or `nil`):

```
    hash_name = "options" if (hash_name == :undefined || hash_name ==
  '' || hash_name == nil)
```

Then, we set the value to return using the `lookupvar` Puppet function, the fully qualified name of the hash variable and its key:

```
value = lookupvar("#{mod_name}::#{hash_name}")["#{option_name}"]
if (lookupvar("#{mod_name}::#{hash_name}").size > 0)
```

If no value is returned then the default value, expected from the second argument passed to the function, is used:

```
value = "#{default_val}" if (value == :undefined || value == '' ||
value == nil)
```

The output of the function is finally calculated value:

```
return value
end
end
```

We can use this function in ERB templates in a similar way:

```
Listen <%= scope.function_options_lookup(['Listen', '127.0.0.1'])%>
```

Let's also see a statement function such as Puppet's core `fail`.

The structure is very similar, we don't have to specify the type of function and each provided argument is collected on a variable called `vals`:

```
Puppet::Parser::Functions::newfunction(:fail, :arity => -1, :doc =>
"Fail with a parse error.") do |vals|
```

If the argument is an array, its members are converted to a space separated string:

```
vals = vals.collect { |s| s.to_s }.join(" ") if vals.is_a? Array
```

Then it simply raises an exception having as description the string with the arguments provided to the function (this is a function that enforces a failure, after all):

```
raise Puppet::ParseError, vals.to_s
end
```

We can use functions for a wide variety of needs:

- Many functions in Puppet Labs `stdlib` module, such as `chomp`, `chop`, `downcase`, `flatten`, `join`, `strip` and so on, reproduce common Ruby functions that manipulate data and make them directly available in Puppet language
- Others are used to validate data types (`validate_array`, `validate_hash`, and `validate_re`)
- We can also use functions to get data from whatever source (a YAML file, a database, a REST interface) and make it directly available in our manifests.

The possibilities are endless, whatever can be possible in Ruby can be made available within our manifests with a custom function.

Custom facts

Facts are the most comfortable kind of variables to work with:

- They are at top scope, therefore easily accessible in every part of our code
- They provide direct and trusted information, being executed on the client
- They are computed: their values are set automatically and we don't have to manage them manually.

Out-of-the-box, depending on Facter version and the underlying OS, they give us:

- Hardware information (`architecture bios_* board* memory* processor virtual`)
- Operating system details (`kernel* osfamily operatingsystem* lsb* sp_* selinux ssh* timezone uptime*`)
- Network configuration (`hostname domain fqdn interfaces ipaddress_* macaddress_* network*`)
- Disks and filesystems (`blockdevice_* filesystems swap*`)
- Puppet related software versions (`facterversion puppetversion ruby*`)

We already use some of these facts in modules to manage the right resources to apply for our operating systems, and we already classify nodes according to their hostname.

There's a lot more that we can do with them, we can create custom facts useful to:

- Classify our nodes according to their functions and locations (using custom facts that might be named `(dc env region role node_number` and so on)
- Determine the version and status of a program (`php_version mysql_cluster glassfish_registered` and so on)
- Return local metrics of any kind (`apache_error_rate, network_connections`)

Just think about any possible command we can run on our servers and how its output might be useful in our manifests to model the resources we provide to our servers.

We can write custom facts in two ways:

- As Ruby code, distributed to clients via `pluginsync` in our modules
- Using the `/etc/facter/facts.d` directory where we can place plain text, JSON, YAML files, or executable scripts

Ruby facts distributed via `pluginsync`

We can create a custom fact editing, in one of our modules, files called `lib/facter/<fact_name>.rb`, for example, a fact named `role` should be placed in a file called `lib/facter/role.rb` and may have content as follows:

```
require 'facter'
```

We require the `facter` class and then we use the `add` method for our custom fact called `role`:

```
Facter.add("role") do
```

We can restrict the execution of this fact only to specific hosts, according to any other facts' values. This is done using the `confine` statement, here based on the `kernel` fact:

```
  confine :kernel => [ 'Linux' , 'SunOS' , 'FreeBSD' , 'Darwin' ]
```

We can also set a maximum number, in seconds, to wait for its execution:

```
    timeout = 10
```

We can even have different facts with the same name and give a weight to each of them; facts with higher weight value are executed first and facts with lower weight are executed only if no value is already returned:

```
has_weight = 40
```

We use the `setcode` method of the `Facter` class to define what our fact does; what is returned from the block of code contained here (between `do` and `end`) is the value of our fact:

```
setcode do
```

We can have access to other facts values with `Facter.value`, in our case, the role value is a simple extrapolation of the `hostname` (basically the `hostname` without numbers). If we have different naming schemes for our nodes and we can deduce their role from their name, we can easily use other Ruby string functions to extrapolate it:

```
  host = Facter.value(:hostname)
  host.gsub(/\d|\W/, "")
end
end
```

Often, the output of a fact is simply the execution of a command; for this case, there is a specific wrapper method called `Facter::Util::Resolution.exec`, which expects the command to execute as a parameter.

The following fact, called `last_run`, simply contains the output of the command `date`:

```
require 'facter'
Facter.add("last_run") do
  confine :kernel => [ 'Linux' , 'SunOS' , 'FreeBSD' , 'Darwin' ]
  setcode do
    Facter::Util::Resolution.exec('date')
  end
end
```

We can have different facts, with different names, on the same Ruby file, and we can also provide **dynamic facts**, for example, the following code creates one different fact, returning the installed version for each package. Here, the confinement according to the `osfamily` is done outside the fact definition:

```
if Facter.osfamily == 'RedHat'
  IO.popen('yum list installed').readlines.collect do |line|
    array = line.split
    Facter.add("#{array[0]}_version") do
      setcode do
```

```
        "#{array[1]}"
      end
    end
  end
end
if Facter.osfamily == 'Debian'
  IO.popen('dpkg -l').readlines.collect do |line|
    array = line.split
    Facter.add("#{array[1]}_version") do
      setcode do
        "#{array[2]}"
      end
    end
  end
end
end
end
```

Remember that to have access, from the shell, to the facts we distribute via `pluginsync`, we need to use the `--puppet (-p)` argument:

```
facter -p
```

Otherwise, we need to set the `FACTERLIB` environment variable pointing to the `lib` directory of the module that provides it, for example:

```
export FACTERLIB=/opt/puppetlabs/puppet/cache/lib/facter
```

External facts in facts.d directory

Puppet Labs's `stdlib` module provides a very powerful addendum to the facts, a feature called **external facts**, which has proven to be so useful to deserve inclusion directly into core Facter since version 1.7.

We can define new facts without even the need to write Ruby code, just by placing files in the `/opt/puppetlabs/facter/facts.d/` or `/etc/facter/facts.d/` directory (`/etc/puppetlabs/facter/facts.d` with Puppet Enterprise and `C:\ProgramData\PuppetLabs\facter\facts.d\` on Windows).

These files can be simple `.txt` files such as `/etc/facter/facts.d/node.txt`, with facts declared with the following syntax:

```
role=webserver
env=prod
```

YAML files such as `/etc/facter/facts.d/node.yaml`, which for the same sample facts would appear like this:

```
---
  role: webserver
  env: prod
```

Also, JSON files such as `/etc/facter/facts.d/node.json` with:

```
{
  'role': 'webserver',
  'env': 'prod'
}
```

We can also place plain commands in any language; on Unix, any executable file present in `/etc/facter/facts.d/` is run and is expected to return the facts' values with an output like this:

```
role=webserver
env=prod
```

On Windows, we can place files with `.com`, `.exe`, `.bat`, `.cmd`, or `.ps1` extensions.

Since Puppet 3.4.0, external facts can also be automatically propagated to clients with `pluginsync`; in this case, the directory synced is `<modulename>/facts.d` (note that this is at the same level of the `lib` directory).

Alternatively, we can use other methods to place our external facts, for example, in post installation scripts during the initial provisioning of a server, or using directly Puppet's file resources, or having them generated by custom scripts or cron jobs.

Custom types and providers

If we had to name a single feature that defines Puppet, it would probably be its approach to the management of systems resources.

The abstraction layer that types and providers provide saves us from worrying about implementations on different operating systems of the resources we want on them.

This is a strong and powerful competitive edge of Puppet, and the thing that makes it even more interesting is the possibility of easily creating custom types and providers and seamlessly distributing them to clients.

Types and providers are the components of Puppet's Resource Abstraction Layer; even if strongly coupled, they do different things:

- Types abstract a physical resource and specify the interface to its management exposing **parameters** and **properties** that allow users to model the resource as desired.
- Providers implement on the system the types' specifications, adapting to different operating systems. They need to be able to query the current status of a resource and to configure it to reflect the expected state.

For each type, there must be at least one provider and each provider may be tied to one and only one type.

Custom types can be placed inside a module in files such as `lib/puppet/type/<type_name>.rb`, and providers are placed in `lib/puppet/provider/<type_name>/<provider_name>.rb`.

Before analyzing a sample piece of code, we will recapitulate what types are about:

- We know that they abstract the resources of a system
- They expose parameters to shape them in the desired state
- They have a title, which must be unique across the catalog
- One of their parameters is `namevar`; if not set explicitly its value is taken from the title

Let's see a sample custom native type, what follows manages the execution of `psql` commands and is from the Puppet Labs' `postgresql` module (<https://github.com/puppetlabs/puppetlabs-postgresql>); we find it in `lib/puppet/type/postgresql_psql.rb`:

```
Puppet::Type.newtype(:postgresql_psql) do
```

A type is created by calling the `newtype` method of the `Puppet::Type` class. We pass the type name, as a symbol, and a block of code with the type's content.

Here, we just have to define the parameters and properties of the type, exactly the ones our users will deal with.

Parameters are set with the `newparam` method, here the `name` parameter is defined with a brief description and is marked as `namevar` with the `isnamevar` method:

```
newparam(:name) do
  desc "An arbitrary tag for your own reference; the name of the
message."

end
```

Every type must have at least one mandatory parameter, the `namevar`, the parameter that will identify each resource among the ones of its type. Each type must have exactly one `namevar`. There are three options to set the `namevar`:

- Parameter `name` is a special case as most types use it as the `namevar`; it automatically becomes the `namevar`. In the previous example, the parameter would be `namevar`.
- Providing the `:namevar => true` argument to the `newparam` call:

```
newparam(:path, :namevar => true) do
  ...
end
```

- Calling the `isnamevar` method inside the `newparam` block:

```
newparam(:path) do
  isnamevar
end
```

Types may have **parameters**, which are instances of the `Puppet::Parameter` class, and **properties**, instances of `Puppet::Property`, which inherits `Puppet::Parameter` and all its methods.

The main difference between a property and a parameter is that a property model is a part of the state of the managed resource (it define a characteristic), whereas a parameter gives information that the provider will use to manage the properties of the resource.

We should be able to discover the status of a resource's property and modify it.

In the type, we define them. In the providers, we query their status and change them.

For example, the built-in type `service` has different arguments: `ensure` and `enable` are properties, all the others are parameters.

The file type has these properties: `content`, `ctime`, `group`, `mode`, `mtime`, `owner`, `seluser`, `selrole`, `seltype`, `selrange`, `target`, and `type`; they represent characteristics of the file resource on the system.

On the other side, its parameters are: `path`, `backup`, `recurse`, `recurselimit`, `replace`, `force`, `ignore`, `links`, `purge`, `sourceselect`, `show_diff`, `source`, `source_permissions`, `checksum`, and `selinux_ignore_defaults`, which allow us to manage the file in various ways, but which are not direct expressions of the characteristics of the file on the system.

A property is set with the `newproperty` method, here is how the `postgresql_psql` type sets the `command` property, which is the SQL query we have to do:

```
newproperty(:command) do
  desc 'The SQL command to execute via psql.'
```

A default value can be defined here. In this case, it is the resource name:

```
defaultto { @resource[:name] }
```

In this specific case, the `sync` method of `Puppet::Property` is redefined to manage this particular case:

```
def sync(refreshing = false)
  if (!@resource.refreshonly? || refreshing)
    super()
  else
    nil
  end
end
```

Other parameters have the same structure:

```
newparam(:db) do
  desc "The name of the database to execute the SQL command
against."
end

newparam(:search_path) do
  desc "The schema search path to use when executing the SQL
command"
end

newparam(:psql_user) do
  desc "The system user account under which the psql command should
be executed."
  defaultto("postgres")
end
[...]
```

The `postgresql_psql` type continues with the definition of other parameters, their description and, where possible, the default values.

If a parameter or a property is required, we set this with the `isrequired` method, we can also validate the input values if we need to force specific data types or values, and normalize them with the `munge` method.

A type can also be made ensurable, that is, have an `ensure` property that can be set to `present` or `absent`, the property is automatically added just by calling the `ensurable` method.

We can also set automatic dependencies for a type; for example, the `exec` native type has an automatic dependency for a user resource that creates the user who is supposed to run the command as, if this is set by its name and not its `uid`, this is how it is done in `lib/puppet/type/exec.rb`:

```
autorequire(:user) do
  # Autorequire users if they are specified by name
  if user = self[:user] and user !~ /^\\d+$/
    user
  end
end
```

We can use such a type in our manifests, here is, for example, how it's used in the `postgresql::server::grant` define of the Puppet Labs' `postgresql` module:

```
$grant_cmd = "GRANT ${_privilege} ON ${_object_type} \"${objectname}\"
TO \"${role}\""

postgresql_psql { $grant_cmd:
  db          => $on_db,
  port        => $port,
  psql_user   => $psql_user,
  psql_group  => $group,
  psql_path   => $psql_path,
  unless      => "SELECT 1 WHERE ${unless_function}('${role}',
'${object_name}', '${unless_privilege}')" ,
  require     => Class['postgresql::server']
}
```

For each type, there must be at least one provider. When the implementation of the resource defined by the type is different according to factors such as the operating system, we may have different providers for a given type.

A provider must be able to query the current state of a resource and eventually configure it according to the desired state, as defined by the parameters we've provided to the type.

We define a provider by calling the `provide` method of `Puppet::Type.type()`; the block passed to it is the content of our provider.

We can restrict a provider to a specific platform with the `confine` method and, in case of alternatives, use the `defaultfor` method to make it the default one.

For example, the `portage` provider of the `package` type has something like:

```
Puppet::Type.type(:package).provide :portage, :parent =>
Puppet::Provider::Package do
  desc "Provides packaging support for Gentoo's portage system."
  has_feature :versionable
  confine :operatingsystem => :gentoo
  defaultfor :operatingsystem => :gentoo
  [...]
end
```

In the preceding example, `confine` has matched a fact value, but it can also be used to check for a file existence, a system's feature, or any piece of code:

```
confine :exists => '/usr/sbin/portage'
confine :feature => :selinux
confine :true => begin
  [Any block of code that enables the provider if returns true]
end
```

Note also the `desc` method, used to set a description of the provider, and the `has_feature` method, used to define the supported features of the relevant type.

The provider has to execute commands on the system. These are defined via the `command` or `optional_command` methods; the latter defines a command, which might not exist on the system and is not required by the provider.

For example, the `useradd` provider of the `user` type has the following commands defined:

```
Puppet::Type.type(:user).provide :useradd, :parent => Puppet::Provider
::NameService::ObjectAdd do
  commands :add => "useradd", :delete => "userdel", :modify =>
"usermod", :password => "change"
  optional_commands :localadd => "luseradd"
end
```

When we define a command, a new method is created; we can use it where needed, passing eventual arguments via an array. The defined command is searched in the path, unless specified with an absolute path.

All the types' property and parameter values are accessible via the `[]` method of the resource object, which are `resource[:uid]` and `resource[:groups]`.

When a type is ensurable, its providers must support the `create`, `exists?` and `destroy` methods, which are used, respectively, to create the resource type, check whether it exists, and remove it.

The `exists?` method, in particular, is at the basis of Puppet idempotence, since it verifies that the resource is in the desired state or needs to be synced.

For example, the `zfs` provider of the `zfs` zone implements these methods running the (previously defined) `zfs` command:

```
def create
  zfs *([:create] + add_properties + [@resource[:name]])
end

def destroy
  zfs(:destroy, @resource[:name])
end

def exists?
  if zfs(:list).split("\n").detect { |line| line.split("\s")[0] == @
resource[:name] }
    true
  else
    false
  end
end
```

For every property of a type, the provider must have methods to read (getter) and modify (setter) its status. These methods have exactly the same name of the property, with the setter ending with an equal symbol (=).

For example, the `ruby` provider of the `postgresql_psql` type we have seen before has these methods to manage the command to execute (here we have removed the implementation code):

```
Puppet::Type.type(:postgresql_psql).provide(:ruby) do

  def command()
    [ Code to check if sql command has to be executed ]
  end

  def command=(val)
    [ Code that executes the sql command ]
  end
end
```

If a property is out of sync, the `setter` method is invoked to configure the system as desired.

Custom report handlers

Puppet can generate data about what happens during a run and we can gather this data in reports. They contain the output of what is executed on the client and details on any action taken during the execution and performance metrics.

Needless to say that we can also extend Puppet reports and deliver them to a variety of destinations: logging systems, database backends, e-mail, chat roots, notification and alerting systems, trouble ticketing software, and web frontends.

Reports may contain the whole output of a Puppet run, a part of them (for example, just the resources that failed) or just the metrics (as it happens with the `rrd` report that graphs key metrics such as Puppet compilation and run times).

We can distribute our custom report handlers via the `pluginsync` functionality too: we just need to place them in the `lib/puppet/reports/<report_name>.rb` path, so that the file name matches the handler name.

James Turnbull, the author of the most popular Puppet books, has written many custom reports for Puppet; here, we analyze the structure of one of his report handlers that sends notifications of failed reports to the PagerDuty service (<https://github.com/jamtur01/puppet-pagerduty>); it should be placed in a module with this path: `lib/puppet/reports/pagerduty.rb`.

First, we need to include some required classes. The Puppet class is always required, others may be required depending on the kind of report:

```
require 'puppet'
require 'json'
require 'yaml'

begin
  require 'redphone/pagerduty'
rescue LoadError => e
  Puppet.info "You need the `redphone` gem to use the PagerDuty
report"
end
```

Next, we call the `register_report` method on the `Puppet::Reports` class, passing to it the handler name, as symbol, and its code in a block:

```
Puppet::Reports.register_report(:pagerduty) do
```

Here, the report handler uses an external configuration file (`/etc/puppet/pagerduty.yaml` (note how we can access Puppet configuration entries with `Puppet.settings[]`), where users can place specific settings (in this case, the PagerDuty API key):

```
config_file = File.join(File.dirname(Puppet.settings[:config]),
"pagerduty.yaml")
raise(Puppet::ParseError, "PagerDuty report config file #{config_
file} not readable") unless File.exist?(config_file)
config = YAML.load_file(config_file)
PAGERDUTY_API = config[:pagerduty_api]
```

We can use the familiar `desc` method to place a description of the report:

```
desc <<-DESC
```

Send notification of failed reports to a PagerDuty service. You will need to create a receiving service in PagerDuty that uses the Generic API, and add the API key to configuration file:

```
DESC
```

All the reporting logic is defined in the `process` method. Here, we can access a lot of information about the report, available as variables of the `self` object; for example, `self.status` contains the status of the Puppet run, `self.logs` all the output text, `self.host` the host where Puppet has been executed. In this case, the `trigger_incident` method of the `Redphone::Pagerduty` class is called and information about a Puppet run is sent if the report status is failed:

```
def process
  if self.status == "failed"
    Puppet.debug "Sending status for #{self.host} to PagerDuty."
    details = Array.new
    self.logs.each do |log|
      details << log
    end
    response = Redphone::Pagerduty.trigger_incident(
      :service_key => PAGERDUTY_API,
      :incident_key => "puppet/#{self.host}",
      :description => "Puppet run for #{self.host} #{self.status} at
#{Time.now.asctime}",
      :details => details
    )
    case response['status']
    when "success"
```



```
        Puppet.debug "Created PagerDuty incident: puppet/#{self.host}"
      else
        Puppet.debug "Failed to create PagerDuty incident:
puppet/#{self.host}"
      end
    end
  end
end
end
```

Custom faces

With the release of Puppet 2.6, a brand new concept was introduced: Puppet faces.

Faces are an API that allow easy creation of new Puppet (sub) commands: whenever we execute Puppet, we specify at least one command, which provides access to the functionalities of its subsystems.

The most common commands are `agent`, `apply`, `master`, and `cert` and have existed for a long time but there are a lot more (we can see their full list with `puppet help`) and most of them are defined via the faces API.

As you can guess, we can easily add new faces and therefore, new subcommands to the Puppet executable just by placing some files in a module of ours.

The typical synopsis of a face reflects the Puppet command's one:

```
puppet [FACE] [ACTION] [ARGUMENTS] [OPTIONS]
```

Where `[FACE]` is the Puppet subcommand to be executed, `[ACTION]` is the face's action we want to invoke, `[ARGUMENTS]` is its arguments, and `[OPTIONS]` is general Puppet options.

To create a face, we have to work on two files: `lib/puppet/application/<face_name>.rb` and `lib/puppet/face/<face_name>.rb`. The code in the `application` directory simply adds the subcommand to Puppet extending the `Puppet::Application::FaceBase` class; the code in the `face` directory manages all its logic and what to do for each action.

An interesting point to consider when writing and using faces is that we have access to the whole Puppet environment, its indirections and termini, and we can interact with its subsystems via the other faces.

A very neat example of this is the `secret_agent` face, which reproduces as a face what the, much older, `agent` command does; a quick look at the code in `lib/puppet/face/secret_agent.rb`, amended of documentation and marginal code, reveals the basic structure of a face and how other faces can be used:

```
require 'puppet/face'
Puppet::Face.define(:secret_agent, '0.0.1') do
  action(:synchronize) do
    default
    summary "Run secret_agent once."
  [...]
  when_invoked do |options|
    Puppet::Face[:plugin, '0.0.1'].download
    Puppet::Face[:facts, '0.0.1'].upload
    Puppet::Face[:catalog, '0.0.1'].download
    report = Puppet::Face[:catalog, '0.0.1'].apply
    Puppet::Face[:report, '0.0.1'].submit(report)
    return report
  end
end
end
```

The `Puppet::Face` class exposes various methods. Some of them are used to provide documentation, both for the command line and the help pages: `summary`, `arguments`, `license`, `copyright`, `author`, `notes`, and `examples`. For example, the module face uses these methods to describe what it does in Puppet's core at `lib/puppet/face/module.rb`:

```
require 'puppet/face'
require 'puppet/module_tool'
require 'puppet/util/colors'

Puppet::Face.define(:module, '1.0.0') do
  extend Puppet::Util::Colors

  copyright "Puppet Labs", 2012
  license "Apache 2 license; see COPYING"

  summary "Creates, installs and searches for modules on the Puppet
Forge."
  description <<-EOT
```

This subcommand can find, install and manage modules from the Puppet Forge, which is a repository of user-contributed Puppet code. It can also generate empty modules, and prepare locally developed modules for release on the Forge:

```
EOT
  display_global_options "environment", "modulepath"
end
```

The `action` method is invoked for each action of a face. Here, we pass the action name as a symbol and a block of code, which implements our action using various other methods:

- The methods used for documentation and inline help are `description`, `summary`, and `returns`
- The methods used to manage the parameters used in the command line are `option` and `arguments`
- The methods used to implement specific actions: `when_invoked` (its return value is the output of the command) and `when_rendering`

Let's see the implementation of the `install` action of the `module` face. The following code is in the `lib/puppet/face/module/install.rb` file; it's possible to add the code for each action in separate files as in this case, or on the main face file.

We are dealing with a Ruby class that may require other classes:

```
require 'puppet/forged'
require 'puppet/module_tool/install_directory'
require 'pathname'
```

This is followed by the face definition and the code applied for the `install` action:

```
Puppet::Face.define(:module, '1.0.0') do
  action(:install) do
```

The description methods are:

```
    summary "Install a module from the Puppet Forge or a release
    archive."

    description <<-EOT
      [...]
    EOT

    returns "Pathname object representing the path to the installed
    module."

    examples <<-'EOT'
      [...]
    EOT
```

Then, the expected arguments and the available options are defined (copied here is only the block relative to the `--target-dir` option, various other are present in the original file and are defined in a similar way):

```
arguments "<name>"

option "--force", "-f" do
  summary "Force overwrite of existing module, if any."
  description <<-EOT
    Force overwrite of existing module, if any.
  EOT
end

option "--target-dir DIR", "-i DIR" do
  summary "The directory into which modules are installed."
  description <<-EOT
    [...]
  EOT
end
```

Then, when the `install` action is called, the `when_invoked` block is executed. Here is where the real work is done, and in this case are mostly called methods from the `Puppet::ModuleTool` class and its subclasses:

```
when_invoked do |name, options|
  Puppet::ModuleTool.set_option_defaults options
  Puppet.notice "Preparing to install into #{options[:target_dir]}
  ..."

  forge = Puppet::Forge.new("PMT", self.version)
  install_dir = Puppet::ModuleTool::InstallDirectory.new(Pathname.
new(options[:target_dir]))
  installer = Puppet::ModuleTool::Applications::Installer.
new(name, forge, install_dir, options)

  installer.run
end
```

This action also invokes the `when_rendering` block to format the console output:

```
when_rendering :console do |return_value, name, options|
  if return_value[:result] == :failure
    Puppet.err(return_value[:error][:multiline])
    exit 1
  else
```

```
        tree = Puppet::ModuleTool.build_tree(return_value[:installed_
modules], return_value[:install_dir])
        return_value[:install_dir] + "\n" +
        Puppet::ModuleTool.format_tree(tree)
    end
end
end
end
```

As it happens for many faces, most of the code is in the face directory. The other component of the face, placed in `lib/puppet/application/module.rb`, is just an extension to the `Puppet::Application::FaceBase` class:

```
require 'puppet/application/face_base'

class Puppet::Application::Module < Puppet::Application::FaceBase
end
```

Summary

This chapter has been entirely dedicated to how we can extend Puppet functionalities writing Ruby code. We reviewed the different areas where Puppet can be customized, from the indirector and its termini to the plugins we can deliver via modules.

We have reviewed the most common plugins: facts, functions, types and providers, reports, and faces, trying to outline the needed code components without delving much into specific implementation details.

The best place to look for samples is the Puppet code itself; under the `lib/puppet` directory, we can find the actual implementation of the core components.

How often we find ourselves working on custom plugins written in Ruby will depend on our needs and skills; we might never need to write any of them, but it is useful to know what they are and the principles behind them.

The scope of this chapter was to provide an overall view in order to be able to find where plugins are placed in a module, know how they integrate into Puppet, and have a high-level view of how they can be implemented.

The next chapter enters in to brand new territory. We are going to explore how we can extend Puppet usage to devices different from the usual operating systems: network equipment, storage devices, and cloud instances.

Believe it or not, we can manage them also with Puppet.

11

Beyond the System

Puppet was designed as a configuration management tool for Unix-like systems. It runs on Linux, Solaris, FreeBSD, OpenBSD, AIX, MacOS and, since Version 2.7.6, also on Windows.

Over the years, however, it became clear that automation in a datacenter must also involve other families of devices: network equipment, storage devices, and virtualization solutions.

The same interest of companies such as Cisco and VMware who are investors and technological partners of Puppet Labs could only facilitate Puppet's steps into these territories. We are already seeing the results of these partnerships, and the vision of a software-defined datacenter is also taking shape under a Puppet-driven perspective.

In this chapter, we will review the current status of the projects that allow us to use Puppet in these categories of devices and technologies:

- Network equipment such as switches, routers, load balancers from Cisco, Juniper, and F5
- Cloud and virtualization with VMware, Amazon, Google, Eucalyptus, and OpenStack
- Storage equipment from NetApp

Puppet on network equipment

The automation of network equipment configuration is a common need; when we provision a new system, besides its own settings we often need to manage switch ports to assign it to the correct VLAN, firewalls to open the relevant ports, and load balancers to add the server to a balanced pool.

It is obvious that the possibility to define the configuration of the whole infrastructure, network included, is a powerful and welcomed point.

There are two main challenges that Puppet faces when it has to deal with network devices. They are as follows:

- **Technical:** This is simply due to the impossibility of having the puppet executable running on the device to be managed.
- **Cultural:** This is because at many places network administrators don't know or use Puppet.

For the technical challenge, there is some good news. Alternative approaches have been taken to manage network equipment of different nature and different vendors with Puppet:

- **Proxy mode:** In our manifests, we declare network-related resource types and apply them to normal nodes, running Linux or another Puppet supported OS. On these servers, the relevant providers execute local commands that interact with remote network devices and configure them as needed. Generally, how this can be done depends on the available configuration methods:
 - **Telnet or SSH connections:** These connections are made to the device and from there, local commands are executed to check the status of a resource (interface, VLAN, pool member, and so on) and eventually to modify it using the local CLI syntax.
 - **Web API:** Some devices expose a web interface to their configuration and allow remote management. On the Puppet proxy node, providers make remote connections to these web API to check and sync the status of resources.
 - **SNMP:** Most of the network devices have a SNMP interface and this can be used for their remote management. Even if I am not aware of any module using this approach, this is theoretically possible.
- **Native mode:** Some network devices can run Puppet natively. They may be based on Linux, FreeBSD, and therefore can potentially host the needed Puppet stack. The fruits of Puppet Labs' partnerships with other vendors are providing good results: Cisco Nexus 9000 switches with Cisco NX-OS can run Puppet natively in a dedicated Linux container and Juniper provides a native Puppet package for its Junos OS.

Besides the technical challenges, for which there are some solutions but still much to do, there are **cultural** and **operational** issues to deal with.

In many places, network and system administrators are of different breeds; they operate in different groups and are responsible for their infrastructures, using their own instrumentation.

Puppet's programmatic approach to configuration, which is likely to be pushed by `sysadmins`, might not be well accepted by the network people, who are probably less obsessed by automation and more used to deal with static configurations.

Here is where DevOps culture may make a difference. There is the need to automate, and there are the tools, solution is collaboration, sharing of responsibilities, and good common sense.

Puppet users need just basic management of network devices, not their whole configurations; most of the time it is a matter of setting parameters and vlans on switch interfaces.

Many products provide authorization profiles, which can limit users' permissions, so a sane compromise can be to allow automatic management only for simple port settings and prevent changes to more global and risky core configurations.

Proxy mode with Puppet device application

Many Puppet features originate from community contributions. One of the most versatile and long-standing contributors is definitively Brice Figureau. When there still wasn't anything around on the topic, he proposed an approach to the network device management, which has been the foundation for the approach based on the proxy mode we mentioned earlier.

In his blog post at <http://puppetlabs.com/blog/puppet-network-device-management>, he introduced the `puppet device` application in Puppet 2.7 to manage external devices where Puppet cannot run natively.

This command uses `/etc/puppetlabs/puppet/device.conf`, by default, as the configuration file. Here, the hostnames of the equipment to manage, their type, and the method to connect to them can be placed. A sample entry may look like the following code:

```
[switch01.example42.lan]
  type cisco
  url ssh://puppet:my_password@switch01.example42.lan/

[router01.example42.lan]
  type cisco
  url telnet://puppet:my_pass@router01.example42.lan/?enable=enablepassword
```


With such a file in place, we can use the `puppet device` command on the host we want to act as proxy for the configuration of remote devices.

The first time this command is executed, it creates certificates for all the devices we have defined in `device.conf`. These certificates have to be signed by the Puppet Master as normal node's certificates.

The implementation provides two core native types: `interface` and `vlan`, with a provider to manage Cisco IOS-based devices. We can execute `puppet describe interface / vlan` for details on their attributes.

To manage switch interfaces (speed and duplex, VLAN, port mode (access/trunk), description, and so on) we can write resources like this:

```
interface { 'FastEthernet 0/1':  
  description => "Server ${server_name}",  
  mode        => access,  
  native_vlan => 1000,  
  duplex      => auto,  
  speed       => auto,  
}
```

To manage router interfaces, we can use the following code:

```
interface { 'Vlan12':  
  ipaddress => [ "192.168.14.14/24", "2001:2674:8C23::1/64" ]  
}
```

To manage VLANs (their ID is the same title) is enough a resource as follows:

```
vlan { '105':  
  description => 'DMZ',  
}
```

These resources can be declared in nodes definitions that match the device names specified in `device.conf`. When `puppet device` is executed, it behaves like `puppet apply`: it retrieves facts from the network device, then retrieves a catalog from the Puppet Master for the locally configured devices and runs it providing a normal transaction report, with the notable difference compared to a normal Puppet run, that the providers that implement the preceding types perform configurations on remote network devices.

On Puppet's core source, there is currently just the provider for Cisco devices and supported transport methods are just `telnet` and `ssh`, but we can find modules that use the same approach and implement it on different devices.

For example, Puppet Labs' F5 module <https://forge.puppetlabs.com/puppetlabs/f5> (Puppet Enterprise is required for installation), introduces several F5 specific resource types, but is based on the network device application and has similar usage patterns. A sample entry in `device.conf` might look like the following code:

```
[f5.example42.lan]
  type f5
  url https://username:password@f5.example42.lan/
```

Note that, in this case, the network device type is `f5` and the access is done via `https`.

A further demonstration of Puppet expandability is a module available at <https://github.com/uniak/puppet-networkdevice> written by two community members, Markus Burger and David Schmitt, which provides wider support for Cisco devices and implements, over the Puppet device application, a new device type (`cisco_ios`). A sample entry in `device.conf` looks like the following:

```
[switch01.example42.lan]
  type cisco_ios
  url sshios://user:password@switch01.example42.lan:22/?$flags
```

The module features a more complete set of resource types to manage different elements of a Cisco IOS configuration (access lists, SNMP configuration, interfaces, VLANs, users, and so on)

Something to consider is that the Puppet agent that normally runs as a service on a node does not implement any device activity. To manage the configured devices on a regular basis we need to place, on the proxy host, a cron job that executes `puppet device`.

Native Puppet on network equipment

A proxy-based approach, based on `puppet device`, has the benefit of letting us manage virtually any device that in some way allows programmatic remote configuration but has some cons, related to scale, authentication management, and the facts that it behaves differently to any other Puppet command.

You can go a step further when Puppet runs natively on the device to be managed and can apply configurations directly. This is an emerging field where we are already seeing some implementations and which will probably grow with the same concept of the software-defined data center.

Cisco onePK

In 2013, Cisco released **onePK**, a Software Development Toolkit that consists of a set of API libraries that allow monitoring and management of different families of Cisco devices and operating systems (IOS / XE, NXOS, and IOS XR), exposing an abstracted interface, which may be used by libraries in different languages.

The family of enterprise switches Nexus 9000 hosts, in a Linux VM container running inside the NXOS, a native Puppet agent which allows the usage of dedicated resource types such as, `cisco_device`, `cisco_interface`, and `cisco_vlan` in a normal Agent / Master setup. We can place the code in a device node as follows:

```
node 'switch01.example42.lan' {
  # Definition of the Device, needed for each device
  cisco_device { 'switch01.example42.lan':
    ensure => present,
  }

  # Configuration of a VLAN on an access interface
  cisco_interface { 'Ethernet1/5':
    switchport => access,
    access_vlan => 1000,
  }

  # Configuration of a VLAN
  cisco_vlan { '1000':
    ensure      => present,
    vlan_name   => 'DMZ',
    state       => active,
  }
}
```

Directly from the device CLI, we can issue commands such as `onep application puppet v0.8 puppet_agent` to run Puppet from the local device, which has its normal certificate and communicates with the Puppet Master as any other node.

The previous resources, when applied on the Linux container where Puppet runs, actually don't operate directly on the switch's configuration, they rather use the onePK presentation API to interface to onePK API infrastructure running on the device.



For more information about Cisco onePK and Puppet refer to this presentation at <http://puppetlabs.com/presentations/managing-cisco-devices-using-puppet>.

Juniper and the netdev_stdlib

Also Juniper Network boasts a deeper approach to Puppet integration. It provides native `jpuppet` packages for its **Junos OS** supported on all releases after 12.3R2. They install on Juniper devices Ruby, the required gems, and Puppet, which runs locally and behaves absolutely as any other client, with its certificates and node definition.

Juniper has also developed two modules:

- A general, open source, abstracted interface to network configurations, `netdev_stdlib` (now under the Puppet Labs' control and can be found at <https://github.com/puppetlabs/puppet-netdev-stdlib>), which contains `netdev_*` types
- A vendor specific module (<https://github.com/Juniper/puppet-netdev-stdlib-junos>), where just the relevant provider for Junos devices is present

The Puppet code for a switch node looks like the following:

```
node 'switch02.example42.lan' {

  # A single netdev_device resource must be present
  netdev_device { $hostname: }

  # Sample configuration of an interface
  netdev_interface { 'ge-0/0/0':
    admin => down,
    mtu    => 2000,
  }

  # Sample configuration of a VLAN
  netdev_vlan { 'vlan102':
    vlan_id      => '102',
    description => 'Public network',
  }

  # Configuration of an access port without VLAN tag
  netdev_l2_interface { 'ge-0/0/0':
    untagged_vlan => Red
  }

  # Configuration of a trunk port with multiple VLAN tags
  # And untagged packets go to 'native VLAN'
  netdev_l2_interface { 'xe-0/0/2':
```

```
        tagged_vlans => [ Red, Green, Blue ],
        untagged_vlan => Yellow
    }

    # Configuration of Link Aggregation ports (bonding)
    netdev_lag { 'ae0':
        links => [ 'ge-0/0/0', 'ge-1/0/0', 'ge-0/0/2', 'ge-1/0/2' ],
        lacp  => active,
        minimum_links => 2
    }
}
```

The idea of the authors is that `netdev_stdlib` might become a standard interface to network devices configurations, with different modules providing support for different vendors.

This approach looks definitively more vendor neutral than Cisco's one based on onePK, and has implementations from **Arista Networks** (<https://github.com/arista-eosplus/puppet-netdev>) and **Mellanox** (<https://github.com/Mellanox/mellanox-netdev-stdlib-mlnxos>).

This means that the same preceding code with `netdev_*` resource types can be used on network devices from different vendors: the power of Puppet's resource abstraction model and the great work of a wonderful community.

Puppet for cloud and virtualization

Puppet is a child of our times: the boom of virtualization and cloud computing have boosted the need and the diffusion of software management tools that can accelerate deployment and the scale of new systems.

Puppet can be used to manage various aspects related to cloud computing and virtualization:

- It can configure virtualization and cloud solutions, such as VMware, OpenStack, and Eucalyptus. This is done with different modules for different operating systems.
- It can provide commands to provision instances on different clouds, such as Amazon AWS, Google Compute Engine, and VMware. This is done with the cloud provisioner module and other community modules.

Let's review the most relevant projects in these fields.

VMware

VMware is a major investor in Puppet Labs and technological collaborations have been done at various levels. Let's see the most interesting projects.

VM provisioning on vCenter and vSphere

Puppet till version 3.8, provides support to manage virtual machine instances using vSphere and vCenter. This is done via a face, which provides the `node_vmware` Puppet sub-command. Once the local environment is configured (for which we suggest you read the documentation on https://docs.puppetlabs.com/pe/3.8/cloudprovisioner_vmware.html), we can create a new VM based on an existing template with the following command:

```
puppet node_vmware create --name=myserver --template="/Datacenters/
Solutions/vm/master_template"
```

We can start and stop an existing VM with these commands:

```
puppet node_vmware start /Datacenters/Solutions/vm/myserver
puppet node_vmware stop /Datacenters/Solutions/vm/myserver
```

vCenter configuration

Puppet Labs and VMware products also interoperate on the setup and configuration of **vCenter**: the VMware application that allows the management of the virtual infrastructure. The module <https://github.com/puppetlabs/puppetlabs-vcenter> can be used to install (on Windows) and configure vCenter.

It provides native resource types to manage objects such as folders (`vc_folder`), datacenters (`vc_datacenter`), clusters (`vc_cluster`), and hosts (`vc_host`).

The Puppet code to manage the installation of vCenter on Windows and the configuration of some basic elements may look like the following:

```
class vcenter {
  media          => 'e:\\',
  jvm_memory_option => 'M',
}

vc_folder { ['/prod':
  ensure => present,
}

vc_datacenter { [ '/prod/uk', '/prod/it' ]:
```

```
    ensure => present,
  }

  vc_cluster { [ '/prod/uk/fe', '/prod/it/fe' ]:
    ensure => present,
  }

  vc_host { '10.42.20.11':
    ensure    => 'present',
    username  => 'root',
    password  => 'password',
    tag       => 'fe',
  }

  vc_host { '10.42.20.12':
    ensure    => 'present',
    username  => 'root',
    password  => 'password',
    tag       => 'fe',
  }
}
```

vSphere virtual machine management with resource types

For more modern versions of the puppet enterprise (from Puppet 3.8), Puppet Labs maintains a module to manage vSphere Virtual Machines; this module requires `rbvmomi` and `hocn` Ruby gems, and can be installed as any other module with:

```
puppet module install puppetlabs-vsphere
```

Refer to the official documentation for more details about its installation:

<https://forge.puppet.com/puppetlabs/vsphere>

Before using it, we need to configure the module to be able to access the vCenter console; it can be configured using environment variables or a configuration file.

The environment variables are:

```
VCENTER_SERVER='host'
VCENTER_USER='username'
VCENTER_PASSWORD='password'
VCENTER_INSECURE='true/false'
VCENTER_SSL='true/false'
VCENTER_PORT='port'
```

And if we want to use the configuration file instead, we must place a file with these values in `/etc/puppetlabs/puppet/vcenter.conf` using this format:

```
vcenter: {
  host: "host"
  user: "username"
  password: "password"
  port: port
  insecure: false
  ssl: false
}
```

Once installed and configured, the `vsphere_vm` resource is available; this allows to manage and list existing vms with `puppet resource`:

```
puppet resource vsphere_vm
```

For example, we could remove a vm with:

```
puppet resource vsphere_vm /opdx1/vm/eng/sample ensure=absent
```

And of course, it's possible to define vms with puppet code:

```
vsphere_vm { '/opdx1/vm/eng/sample':
  ensure => running,
  source => '/opdx1/vm/eng/source',
  memory      => 1024,
  cpus        => 1,
  extra_config => {
    'advanced.setting' => 'value',
  }
}
```

Amazon Web Services

Puppet-based solutions to manage AWS services have been around for some time. There are contributions both from Puppet Labs and the community and they relate to different Amazon services.

Cloud provisioning on AWS

Puppet Labs released a Cloud Provisioner module that provides faces to manage instances on AWS and Google Compute Engine. We can install it with:

```
puppet module install puppetlabs-cloud_provisioner
```


A new `node_aws` face is provided and it allows operations on AWS instances. They are performed via Fog, a Ruby cloud services library, and need some prerequisites. We can install them with:

```
gem install fog
gem install guid
```



On Puppet Enterprise, all the cloud provisioner tools and their dependencies can be easily installed on the Puppet Master or any other node directly during Puppet installation.

In order to be able to interface to AWS services, we have to generate Access Credentials from the AWS Management Console. If we use the, now recommended **AWS Identity and Access Management (IAM)** interface, remember to set at least a Power User policy to the user for which access keys are created. For more details, check out <https://aws.amazon.com/iam/>. We can place the access key ID and secret access key in `~/.fog`, which is the configuration file of Fog:

```
:default:
  aws_access_key_id: AKIAILAJ3HL2DQC37HZA
  aws_secret_access_key: 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
```

Once done, we can interact with AWS. To see the `ssh` key pair names we have on AWS, we run the following command:

```
puppet node_aws list_keynames
```

To create a new instance, we can execute the following command:

```
puppet node_aws create --type t1.micro --image ami-2f726546 --keyname
my_key
```

We have specified the instance type, the **Amazon Machine Image (AMI)** to be used and the SSH key we want to use to connect to it.

The output of the command reports the hostname of the newly created instance so that we can SSH to it with a command such as the following one (if you have issues with connecting via SSH, review your instance's security group and verify that inbound SSH traffic is permitted on the AWS console):

```
ssh -i .ssh/aws_id_rsa root@ec2-54-81-87-78.compute-1.amazonaws.com
```

The list of all the instances (both stopped and running) that we use are:

```
puppet node_aws list
```

To destroy a running instance (it is going to be wiped off forever):

```
puppet node_aws terminate ec2-54-81-87-78.compute-1.amazonaws.com
```

We can specify the AWS region where instances are created with the `--region` option (the default value is `us-east-1`).

AWS provisioning and configuration with resource types

Since Puppet 3.8, the recommended way of managing AWS resources with Puppet is by using the official AWS module (<https://github.com/puppetlabs/puppetlabs-aws>), this can manage AWS services to configure cloud infrastructure.

To access AWS services, Puppet needs the SDK and the credentials. To install the SDK, run:

```
gem install aws-sdk-core retries
```

You may need to execute a different `gem` depending on your installation and how you are going to run AWS management commands. For Puppet enterprise and starting on Puppet 4, `gem` is in `/opt/puppetlabs/puppet/bin/gem`; if the code is going to be executed from a `puppetserver`, `gem` has to be invoked as `/opt/puppetlabs/bin/puppetserver gem`, and the server needs to be reinstalled to see the new gems.

Credentials have to be set as environment variables:

```
export AWS_ACCESS_KEY_ID=your_access_key_id
export AWS_SECRET_ACCESS_KEY=your_secret_access_key
export AWS_REGION=region
```

Otherwise, in a file at `~/.aws/credentials`:

```
[default]
aws_access_key_id = your_access_key_id
aws_secret_access_key = your_secret_access_key
aws_region = region
```

Finally, install the Puppet module itself with the following command:

```
puppet module install puppetlabs-aws
```

Once installed and with the configured credentials, it can be used as any other resource, directly from code or using the command line. As an example, a new instance can be launched with this Puppet code:

```
ec2_instance { 'instance-name':  
  ensure      => present,,  
  image_id     => 'ami-123456',  
  instance_type => 't1.micro',  
  key_name     => 'key-name',  
}
```

Otherwise, with this equivalent command:

```
puppet resource ec2_instance instance-name \  
  ensure=present \  
  image_id=ami-123456 \  
  instance_type=t1.micro \  
  key_name=key-name
```

There are types defined for multiple resources as can be seen in the reference documentation at <https://forge.puppetlabs.com/puppetlabs/aws#reference>.

Managing CloudFormation

We have seen how Puppet functionalities can be extended with modules, either by providing resource types that enrich the language or additional faces that add actions to the application.

One of these extra faces is provided by the Puppet Labs' Cloud Formation module, <https://github.com/puppetlabs/puppetlabs-cloudformation>. It adds the `puppet cloudformation` subcommand, which can be used to deploy a whole Puppet Enterprise stack via Amazon's Cloud Formation service.

The module, besides installing a Master based on Puppet Enterprise, configures various AWS resources (security groups, IAM users and ec2 instances) and Puppet specific components (modules, dashboard groups, and agents).

Cloud provisioning on Google Compute Engine

With a similar approach to the AWS module, there is another one for Google Compute Engine resources.

It can be found with installation instructions at https://forge.puppetlabs.com/puppetlabs/gce_compute.

Its usage pattern is very similar to the AWS module; it provides a collection of defined types that can be used in Puppet code or using the `puppet resource` command.

Puppet on storage devices

Puppet management of storage devices is still at the early stages, there are not many implementations around but something is moving.

For example, there is Gavin Williams' module to manage NetApp filers:

<https://forge.puppetlabs.com/puppetlabs/netapp>.

In addition, this module is based on `puppet device`, the configuration being something like this:

```
[netapp.example42.lan]
  type netapp
  url https://root:password@netapp.example42.lan
```

The module provides native types to manage volumes (`netapp_volume`), NFS shares (`netapp_export`), users (`netapp_user`), snap mirrors (`netapp_snapmirror`), and other configuration items. It also provides a defined resource (`netapp::vqe`) for easy creation and export of a volume.

Puppet and Docker

With the popularization of containerization technologies, new ways of approaching services provisioning have started to become popular; these technologies are based in the features of operating systems to start processes on the same kernel, but with isolated resources.

If we compare with virtualization technologies, virtual machines are generally started as full operating systems that have access to an emulated hardware stack, this emulated stack introduces some performance penalties, as some translations are needed so the operations in the virtual machine can reach the physical hardware. These penalties do not exist in containerization, because containers are directly executed on the host kernel and over the physical hardware. Isolation in containers happens at the level of operating system resources.

Before talking about the implications containers have for systems provisioning, let's see some examples of the isolation technologies the Linux kernel offers to containers:

- **Control Groups** (more often known as cgroups) are used to create groups of processes that have access quotas for CPU and memory usage.
- **Capabilities** are the set of privileges a traditional full-privileged root user would have. Each process has some bitmasks that indicate which one of these privileges it has. By default, a process started by root users has all privileges, and a process started by another user has no privileges. With capabilities, we have more fine-grained control; we can remove specific privileges from root processes or give privileges to processes started by normal users. In containers, it allows you to make the processes believe they are being run by the root, while at the end, they only have a certain set of privileges. This is a great tool for security, as it helps reduce the surface of possible attacks.
- **Namespaces** are used to create independent collections of resources or elements in the state of the kernel; after defining them, processes can be attached to these namespaces so they can only access the resources in these namespaces. For example:
 - Network namespaces can have different sets of physical or virtual network interfaces, a process in a network namespace can only operate with the interfaces available there. A process in a namespace with just the local interface wouldn't have access to the network even if the host has.
 - Process namespaces create hierarchical views of processes, so processes started in a namespace can only see other descendant processes of the first process in this namespace.
 - There are also other namespaces for inter process communication, mount points, users, system identifiers (hostname and domain), and control groups.

Notice that all of these technologies can be (and indeed are) used by normal processes, but they are the ones that allow containers to be possible.

In general terms, to start a container these steps are followed:

1. An image is obtained and copied if needed. An image can basically be a tarred filesystem and it should contain the executable file we want to run, and all its dependencies.
2. A set of systems resources as cgroups, namespaces, network configurations, or mount points are set up for container use.
3. A process is started by running an executable file located in the base image with the assigned resources created in the previous step.

Docker (<https://www.docker.com/>), probably the toolbox for containerization that has contributed more to popularize these technologies, offers a simple way to package a service in an image with all its dependencies and deploy it the same way in different infrastructure, from developer environments in laptops or the cloud, to production.

Docker creates containers with the principle of immutability: once they are created, they can only be used as they were intended at the moment of building them, and any change implies rebuilding a new container and replacing the running ones with new instances.

It also helps to think in services more than thinking on machines. Ideally, container hosts would be as simple as possible to just run containers, and any service deployed should be fully containerized with their dependencies in an immutable container.

These principles of simplicity in provisioning and immutable services seem opposed to the capacity to propagate changes even on complex deployments of configuration management tools such as Puppet. So, what space is left for these tools in deployments based on containers?

There are some areas where Puppet can still shine in these deployments:

- **Hosts provisioning:** At the end, containers need to be run somewhere, when deploying container hosts, Puppet can be a good option to install the container executor. For example, there are some modules to install and configure Docker such as at <https://github.com/garethr/garethr-docker>. We could prepare a host to run dockers with just this line of puppet code: `include 'docker'`. This module can be also used to retrieve images and start containers with them:

```
docker::image { 'ubuntu1604':  
  image_tag => '16.04'  
}  
docker::run { 'helloworld':  
  image    => 'ubuntu1604',  
  command => '/bin/sh -c "while true; do echo hello world; sleep  
1; done"',  
}
```

- **Container builds:** Even when Docker features its own tool to build containers they have an open format, and other tools can be used to generate these artifacts. This is the case with Hashicorp's Packer (<https://www.packer.io>), a tool that automates the creation of images and containers, that between its list of supported **provisioners** includes Puppet in two flavors: server and Masterless. Server mode can be used to provision a container using an existing server, and Masterless can be used to provision using Puppet code directly by running Puppet agent.



There are multiple formats for container images, the Docker one (implemented by packer) is one of them, but there are also efforts to create an open specification that is independent of the tool used to create or run them. This and other related specifications are maintained by the Open Container Initiative (<https://www.opencontainers.org/>), which includes the main actors in this topic.

- **Other system integrations:** Depending on our deployment, we may require some specific configurations around containerized services such as cloud resources, DNS, networking, or remote volumes. As we have seen in this book, Puppet can be used to configure all these things.

Summary

In this chapter, we have explored less traditional territories for Puppet. We have gone beyond server operating systems and have seen how it is possible to also manage network and storage equipment, and how Puppet can help in working with the cloud and with containers.

We have seen that there are two general approaches to the management of devices: the proxy one, which is mostly implemented by the `puppet device` application and has a specific operational approach, and the native one, where Puppet runs directly on the managed devices, where it behaves like a normal node.

We have also reviewed the modules available to manage virtualization and cloud-related setups. Some of them configure normal resources on a system, others expand the Puppet application to allow creation and interaction with cloud instances.

Puppet is well placed for the future challenges that a software defined data center involves, but its evolution is an ongoing process on many fields.

In the next chapter, we are going to explore how Puppet is evolving and what we can expect from the next versions.

12

Future Puppet

Time is relative.

My future is your present.

These two lines started this chapter in the first edition of this book, and as expected, lots of things have changed since then. Most of the features previously seen as future are already present and some of them are well consolidated. And, as with the first edition, some of the features we'll review here will be not the future, but the present of some of the readers of this book.

While writing this second edition, Puppet 4 has become a reality and we may not be seeing so many revolutionary changes and experimental features as we saw with the latest releases. The roadmap to the next Puppet releases aims more at improvements in existing features and to continue working on performance and on a more optimized codebase. These changes include full code rewrites to other languages, and changes in network protocols.

Puppet also has to respond to how the computing world is changing; everything is in the cloud now and containers are present everywhere, also, other operating systems as Microsoft Windows require the same level of automation as the ones found in Linux and Puppet aspires to be a reference in these platforms too.

In this chapter, we are going to explore the following topics:

- Changes in serialization formats, **MessagePack** and the replacement of PSON by **JSON**.
- **Direct Puppet and cached mode**, to avoid unnecessary Puppet runs by improved changes detection and improvements in protocol.
- **File sync**, the new solution to deploy code in Puppet enterprise.

- Other expected changes as Puppet CA reimplementations, some features for better package management support, or the future of Puppet faces.
- Beyond Puppet 4. Even if it's probably too soon to talk about this, we will explore what seems to be in the roadmap to **Puppet 5**.

Changing the serialization format

Puppet server and clients exchange a remarkable amount of data: facts, catalogs, and reports. All this data has to be serialized, that is, converted into a format that can be stored to a file, managed in a memory buffer, and sent on a network connection.

There are different serialization formats: Puppet, during its years of existence, has used XML-RPC, YAML, and PSON (a custom variation of JSON that allows inclusion of binary objects), the latter being the currently preferred choice.

PSON has some problems: it was for some time pure JSON, but then it evolved separately to be adapted to Puppet convenience, one of the main differences is that JSON is restricted to UTF-8, while PSON accepts any encoding. This change was introduced to allow binary data to be sent as the content of files, but it also introduced the problem of losing control over the encoding Puppet code has to support.

Currently, another protocol supported by Puppet and maintained by the community is MessagePack. It's a binary format that is more compact and efficient (some tests suggest that it can be 50 percent more compact than JSON), so it reduces both the quantity of data to transmit over the wire and the computational resources needed to manage it.

A simple test shows how an array such as `['one' , 'two' , 'three']` can look in different formats. Expressed in the YAML format, this becomes (24 bytes, newlines included):

```
---
- one
- two
- three
```

In the JSON format, this becomes (21 bytes):

```
["one", "two", "three"]
```

In MessagePack, this becomes (15 bytes (a string such as `\x93` represents a single byte in hexadecimal notation):

```
\x93\xA3one\xA3two\xA5three
```

If we consider that Puppet is constantly making serialization and deserialization of data, it's clear that the introduction of such a format may deliver great performance benefits.

To enable MessagePack serialization, we need to install the `msgpack` gem on both clients and server:

```
gem install msgpack
```

Remember to use the appropriate `gem` command depending on your installation. And set it, in the `[main]` section of their `puppet.conf`:

```
preferred_serialization_format = msgpack
```

Clients where `msgpack` is not supported can keep the default PSON format and continue to operate normally. But although on the way to Puppet 3 and 4 MessagePack looked like the right solution, as it uses binary representation, it couldn't help with the encoding problems.

On the other hand, the well-known and universally supported JSON format requires everything to be valid Unicode (PSON is represented with 8-bit ASCII), which can help a lot with this problem. It's not as fast or compact as MessagePack, but it has multiple libraries that can be tested to see which one provides a better performance for Puppet needs.

Puppet also uses YAML to store some data and reports on server and clients; the migration plan to JSON also includes the deprecation of these files. For the migration to new formats, Puppet will keep backwards compatibility with PSON by now, but decoding and encoding everything as JSON from version 5.

Direct Puppet

The Direct Puppet initiative aims to make some improvements in how clients and servers communicate. Most of the efforts will be focused on having more control on how and when catalogs are recompiled, trying to do it just when needed.

This initiative includes a change in the protocol that will make it more efficient. Instead of asking the server for the catalog, the client will have the initiative of sending (before anything) an identifier of the last catalog executed. The communication will follow these steps:

1. Client agent sends last executed `catalog_id` to the server. As part of the direct Puppet initiative, catalogs are intended to change only if a new version of the catalog is released, that means that while the same code is deployed, the same catalog will never be computed for the same node (even if facts change), so Puppet will be able to uniquely identify it.
2. Puppet server checks if this `catalog_id` corresponds to the currently deployed code, if it is, it doesn't need to compile it again; if it isn't, then it has to be compiled as usual, but a new `catalog_id` is also created.
3. Puppet server answers to the client agent; at this step the agent knows what catalog has to be applied. Note that if it was the cached catalog, the communication until now has been way faster.
4. Local files are checked to see whether they match the expected state. But, instead of asking the master to compute all the files and templates and compare them with the local content, a hash is stored for any of the files in the static cached catalog, so the agent just has to compute the hash of the local files and request for the files that do not match with the hash in the catalog.
5. The server retrieves the requested files from its cache.

The main idea behind these changes is to detect as early and efficiently as possible whether some work needs to be done by the master. It will also increase control on when the catalogs and files are computed and built. The only moment that new catalogs will be required will be when new code is deployed; it is a way to give administrators the control to know when changes will be applied. Changes in facts won't trigger new compilations, as it would go against the decision of the administrator to apply changes; this also helps to assume that the same code will always generate not only the same catalog but also the same files.

File sync

With static catalogs, new possibilities for code deployment and file caching appear.

When having multiple Puppet masters serving the same code, we may face the problem of not being sure whether the code is synchronized between them, and in most of the cases, we can even be sure that the code is going to be unsynced at least while we release it, as not all files change at once in all servers. The result of code compilation while the code is being updated is unpredictable.

File sync provides mechanisms to improve how code is deployed:

- It does atomic updates, it allows to update all servers at once while they serve requests
- It ensures that the code has been deployed completely and correctly before starting to serve new requests based on it
- It allows to know exactly what code generated a catalog
- Cached catalogs can be safely invalidated

It basically works by requesting a **master of master** for confirmation to know if code has changed and is transparent for agents.

When code is deployed, these steps are followed:

- Code is deployed to a staging area
- An HTTP POST request is sent to the new endpoint `/file-sync/v1/publish` in Puppet master to start publishing the code
- When masters have to compile, they ask the master of masters for the new code; every request processed after the moment the notification is sent will use this code

A new code manager will also be released to support this new way to release Puppet code, but in any case, it will be very easy for any deployment tool to implement file sync, as it will only need to push the code to a known path and send the request to the Master to start publishing the new code.

In principle, these features will only be available in Puppet Enterprise.

Other changes

There are multiple changes incoming, some of them are already being introduced in the latest Puppet releases.

Certificate authority in Clojure

Certificate authority is being rewritten to Clojure and it will be directly executed by Trapperkeeper. This implementation will not use any of the Ruby code, but it will keep backwards compatibility with older versions. Both implementations will be kept in parallel till the new implementation is fully functional. Some of the features expected are as follows:

- Management unified in a single command
- Improved support to cloud environments, with facilities to make it easier to authorize and remove nodes
- CA completely separated from the master, what can help in high availability scenarios

This reimplementaion will provide a more efficient and more maintainable service, in the line of most of the changes we'll see in the next releases of Puppet.

Package management

On package management, we can also expect improvements in the near future. We'll also see changes in the general behavior of the package resource and better support for some types of packages.

One of the most anticipated changes in this area is the possibility of installing multiple packages at once. There are some scenarios where atomicity in the installation or replacement of multiple packages would be welcomed. Also, most package managers support the installation of multiple packages in a single command, which used to be more efficient and would make performance improvements in Puppet. One of the proposed implementations requires a batch processing subsystem that would group the execution of multiple resources of the same type and provider. We might see some advances on this in the near future.

Another expected feature in this area is the support for PIP and `virtualenv` to manage Python dependencies, it may need to add new providers for that.

Windows provider will also receive support for `misp` and `msu` packages, as well as other improvements.

Changes in faces

There are some changes that are happening in the use of faces in the Puppet ecosystem.

Lots of faces are going to be deprecated or heavily modified in future versions:

- The `face` file is going to be removed as it doesn't provide much functionality over `filebuckets`, and which one must be used is confusing; `filebuckets` documentation and support will be improved.
- Faces related to certificates as `ca`, `cert`, `certificate`, `certificate_request`, `certificate_revocation_list`, and `key` are confusing for lots of users, who in most cases just need the functionality provided by `puppet cert`. There are open discussions about how to reorganize these faces.
- There are doubts about other faces such as `status` or `report` that might be useful, but are difficult to use or understand in the current implementations. Changes or deprecations will probably be seen in these faces.
- In previous chapters we saw that faces included in older modules for cloud management are not being migrated to newer implementations, that dedicate their efforts to have adequate resource types.

And in general, slowly, the use of faces is being discouraged in favor of using Puppet more as a library when trying to extend configuration. This makes it easier to have a consistent API that can be used by the community to create other tools, instead of having a big and unmaintainable collection of faces.

Beyond Puppet 4.x

The future of Puppet is probably going to be determined by its platform changes: on one side, the migration of server and network code to its Clojure platform, and on the other, the migration of client tooling and hot-spots to C++.

Consolidating Clojure implementations will give robustness, better performance, and maintainability to server code. With a micro services approach, each component will be developed in the stack that is most suitable for its specific mission. In Puppet Conf 2015, Peter Heune presented a prototype of an implementation of a compiler compatible with Puppet 4 written in C++. In the demoed examples, new implementation compiled pure Puppet code (Ruby-defined types and functions are not supported yet), orders of magnitude faster than the Ruby one, and as he showed, it also did very good memory management.

And it does just one very specific task, it's just a compiler, it converts Puppet code in to a catalog that can be cached, directly applied by some kind of executor in a Masterless setup, or sent by the network on request by other components. It is just a piece that could be plugged in to the Puppet master Trapperkeeper to offer much better compilation times, while the rest of the functionality is given by the **glue** components provided by the framework.

It's just an example, but it gives an idea of what we can expect in terms of scalability and performance in future Puppet versions.



Puppet Conf talks are usually available on the Puppet Labs website, and the demo comparing Ruby and C++ implementations is at <https://puppet.com/presentations/keynote-product-feature-demo>. The C++ prototype and the information about the features it supports was published and is available at <https://github.com/puppetlabs/puppetcpp>.

Migrating client code to C++ will also allow unexpected uses of Puppet's components. We already enjoy the new implementation of Factor 3, much faster, lightweight, and with more features. A full reimplement of the Puppet agent would bring Puppet to any kind of device. There are multiple cases where installing a full Ruby stack can be seen as a great drawback, as in containers or in embedded devices. But having a set of efficient, small, and modular tools that can help with different provisioning tasks would probably be a great thing.

Summary

The world of systems administration changes, and Puppet does too. It appeared to us to be a very powerful tool that helps us in day-to-day operations, making it easier and comprehensible to provision and maintain infrastructures composed by either a few or hundreds of nodes.

In this chapter, we saw how this tool is evolving in several ambits. We saw on one side new features that will be included in the language and in the APIs for simplicity and to help us write more efficient catalogs. We saw how even best practices change: something that looked like a very good idea to extend Puppet, as was the case of the faces, now is looked on as something difficult to maintain, and new recommendations are given. We also saw that some resources as packages could be managed in different and better ways.

On the other side, at a more internal level, we saw that some parts are changing to give better support and scalability to any kind of deployment. The new CA, the direct Puppet initiative, and plugin sync will change the way Puppet infrastructure is deployed and scaled.

In the near future, we'll see how Puppet adapts to new paradigms as it did with the cloud. Now the disruption caused by the popularization of containers is changing everything. Minimalistic computers and operating systems will soon require the same provisioning tools we have today in general purpose servers. Puppet, with its new and more modular implementations in compiled languages, will probably offer us tools for these newer worlds.

Index

Symbols

- `/aggregated-event-counts endpoint` 86
- `/catalogs endpoint` 83
- `/event-counts endpoint` 85
- `/events endpoint` 84
- `/fact-names endpoint` 83
- `/facts endpoint` 81
- `/metrics endpoint` 83
- `/nodes endpoint` 82
- `/reports endpoint` 84
- `/resources endpoint` 81
- `/server-time endpoint` 86
- `/version endpoint` 86

A

- Abstract Syntax Tree (AST)** 228
- agents** 5
- allowed characters** 28
- Amazon Machine Image (AMI)** 268
- Amazon Web Services**
 - about 267
 - Cloud provisioning 267, 268
- anchor pattern** 121
- apply command** 202
- Arista Networks**
 - reference 264
- augeas**
 - URL 127
- automatic parameter lookup** 51
- automation tasks, existing infrastructure**
 - application deployments, refining 164
 - common systems configurations,
 - automating 164

- integration 165
- monitoring infrastructure, automating 165
- roles, automating 164
- servers deployment, automating 163
- AWS Identity and Access Management (IAM) interface**
 - reference 268
- AWS module**
 - reference 269
- AWS provisioning**
 - about 269
 - CloudFormation, managing 270

B

- backend specific settings, hiera.yaml configuration file**
 - `:datadir` 42
 - `:datasource` 42
- Beaker**
 - about 192-194
 - URL 192
- beaker-rspec**
 - URL 192
 - using 192-194
- BitBucket**
 - URL 187
- built-in variables** 23

C

- cached mode 275
- caching 218
- `ca.conf` file 7
- certificate authority, in Clojure 280

- Certificate Revocation List (CRL)** 212
- certificates**
 - managing 211-213
- Certificate Signing Requests (CSR)** 212
- classes**
 - about 4, 109
 - declaring 13-15
 - defining 13, 99
 - defining, for each node 94, 95
- class inheritance** 16
- class names** 120
- class parameters** 116, 117
- Clojure** 209
- CloudFormation**
 - about 270
 - reference 270
- Cloud provisioning**
 - on Google Compute Engine 270
- code**
 - data separation 103, 104
- code and data**
 - managing 215
- code management**
 - with Git 182, 183
- code optimization** 223, 224
- code review**
 - about 186
 - Gerrit 186
 - online resources, for peer review 187
- command line**
 - used, for querying puppetdbquery module 87
- Command/Query Responsibility Separation (CQRS)** 76
- commands**
 - about 65
 - deactivate node 80
 - replace catalog 79
 - replace facts 79
 - store report 80
 - using 79, 80
- comparison operators** 29
- component (application) modules** 139, 140
- components** 4, 5, 91, 92
- components, Puppet architecture**
 - about 92, 93
 - classes, defining 99
 - classes, defining for each node 94, 95
 - configuration files, defining for each node 98
 - custom resources, defining 99
 - parameters, defining for each node 95-97
- conditionals** 28, 29, 117
- configuration directory** 130
- configuration files**
 - defining, for each node 98
 - placing, in custom fileserver 99
 - placing, in Hiera 98
 - placing, in shared modules 98
 - placing, in site module(s) 98
- configuration hash patterns**
 - managing 128, 130
- configuration management**
 - importance 3, 4
- configuration, PuppetDB**
 - configuration settings 68-73
 - init script configuration 68
 - logging configuration 73
 - on Puppet Master 73, 74
- container builds** 274
- Continuous Integration (CI) tool**
 - about 197
 - Jenkins 199
 - Travis 198, 199
- C++ prototype**
 - reference link 282
- custom faces** 252-255
- custom facts**
 - about 239
 - external facts 242, 243
 - Ruby facts 240, 241
- custom fileserver**
 - configuration files, placing 99
- custom functions**
 - about 236
 - defining 236-238
- custom providers** 243-249
- custom report handlers** 250, 251
- custom resources**
 - defining 99
- custom resources, locations**
 - Hiera 99
 - shared modules 99
 - site module(s) 99

custom types 243-249

D

dashboards

- about 74
- performance dashboard 74, 75
- Puppetboard 76

data binding 14, 51, 97, 235

data separation

- about 92
- from code 103, 104
- in modules 119
- reference link 119

Dead Letter Office 69

default Puppet Master 203, 204

defined resource type 15

defined types 15

dependencies

- managing 25, 26, 132, 133

Direct Puppet

- about 278
- file sync 279

Distinguished Name (DN) 214

DNS SRV records 216

Docker

- about 273
- URL 273

Domain Specific Language (DSL) 4

dynamic facts 241

dynamic scoping 25

E

Eclipse 180

ENC

- extendibility 230
- user variables 22

encryption plugins

- about 55
- hiera-gpg 55-57

environments, Git workflows

- branch based automatic environments 185
- reference link 185
- simplified developer workdir environments 185

EPEL 5

existing infrastructure

- coding 167-169
- modifications, applying 170, 171
- node migration 156
- node update 156
- planning, for automation 161, 162
- priorities, defining 162-165
- refactoring 160
- solutions, evaluating 166, 167
- step-by-step process,
 - for automation 160, 161
- tasks, for automation 163

exported resources

- about 31, 64
- virtual resources 32

expressions combinations 31

external components

- ENC 216
- Exported Resources 217
- Report 217

external facts 19, 242, 243

F

faces

- enhancements 281

factor 4, 234

facts

- about 4, 18, 96
- external facts 19
- system's facts 18

facts.d directory

- external facts 242, 243

facts driven approach

- reference link 112

files

- about 109
- class names 120
- configuration hash patterns,
 - managing 128, 130
- dependencies, managing 132, 133
- extra resources, managing 134-136
- installation options, managing 133, 134
- managing 126-128
- multiple configuration files,
 - managing 130, 131

- restoring, from filebucket 36, 37
- users, managing 132, 133

files-based logic 143

file sync 275, 279

file system

- changes, verifying 219

Foreman

- about 91, 93, 99
- features 100

frontend servers 214

fully qualified domain name (FQDN) 7

G

Gatling

- URL 222

gem 41

general purpose define 130, 131

Geppetto

- about 2, 180
- URL 180

Gerrit

- about 186
- URL 187

GitEnterprise

- URL 187

Git hooks

- about 183
- git/hooks/post-receive 184
- git/hooks/pre-commit 184
- git/hooks/pre-receive 184
- URL 184

GitHub

- URL 187

Git workflows

- about 182
- branches 184
- code management, with Git 182, 183
- environments 184
- Git hooks 183, 184

global.conf file 7

global settings, hiera.yaml configuration file

- :backends 42
- :hierarchy 42
- :logger 42
- :merge_behavior 42

glue components 282

gpg private key 57

grouping classes 105

H

hammer-cli command

- about 101
- URL 101

hash

- dealing with 50, 51

hash merge

- reference link 48

Hiera

- about 2, 40, 93
- configuration files, placing 98
- configuring 40, 41
- custom resources 99
- extendibility 230
- hash, dealing with 50, 51
- installing 40, 41
- Puppet 3 automatic
 - parameter lookup 51, 52
- shared modules 99
- site module(s) 99
- usage patterns, evolving for class
 - parameters 52, 53
- user variables 22
- using 49
- using, as ENC 60

Hiera backend

- about 53, 88, 89
- encryption plugins 55
- hiera-eyaml 57-59
- hiera-file 53-55
- hiera-gpg 55-57
- hiera-http 59
- hiera-mysql 59

Hiera functions, parameters

- second argument 50
- third argument 50

hiera.yaml configuration file

- about 41
- backend specific settings 42
- command line, working with 44-48
- examples 42-44
- global settings 42

hosts provisioning 273

I

idempotence 5

identifying parameters 96

indirector

about 232-236

URL 234

infrastructure automation

approaches, examining 153, 154

existing automated infrastructures,

updating 158-160

existing infrastructures,

automating 155, 156

modifications 171-174

new infrastructures, creating 154, 155

potential scenarios, examining 153, 154

Puppet friendly infrastructure,

managing 174, 175

Puppet-friendly software, using 175, 176

infrastructure logic 169

infrastructure optimizations

about 217

caching 218

file system changes, verifying 219

Puppet run times, distributing 218, 219

stored configs, scaling 219-221

traffic compression 217

inventory service 64

isolation technologies

Capabilities 272

Control Groups 272

Namespaces 272

iteration

about 30

expressions combinations 31

in operator 31

J

Jenkins 199

Jenkins plugins

GitHub / Gerrit 199

RVM / rbenv 199

ssh 199

URL 199

Vagrant 199

JSON 275

Juniper

about 263

modules 263

Junos OS 263

K

Kermit 3

L

lambdas 30

LDAP

about 93

URL 93

librarian-puppet

about 181

Puppet code, deploying with 195

URL 195

load balancing

options 215, 216

M

masterless Puppet 216, 217

Mcollective 2

Mellanox

reference 264

MessagePack 275

meta parameters 25

mod_passenger 204

mod_rails 204

modules

about 4, 33, 115

auto loading 33, 35

data separation 119

path 33, 35

templates 35, 36

Mozilla SSL Configuration Generator

URL 207

multi-master scaling

about 210

certificates, managing 211-213

code and data, managing 215

SSL termination, managing 213-215

multiple configuration files

managing 130, 131

N

native Puppet, on network equipment

- about 261
- Cisco onePK 262
- Juniper, and netdev_stdlib 263, 264

native resources

- reference link 10

NetApp filers

- reference 271

node

- about 4
- classes, defining 94, 95
- configuration files, defining 98
- parameters, defining 95-97

node-less layout 94

nodeless setup 13

node migration

- about 156, 157
- advantages 156, 157

node-purge-ttl settings 71

nodes' classification 94

node-ttl settings 71

node update 157, 158

Nova module

- URL 129

O

official OpenStack module

- about 140
- URL 140
- using 140

Open Container Initiative

- URL 274

OpenStack component 139

OpenStack example

- about 139
- component (application) modules 139, 140
- official OpenStack module, using 140
- OpenStack infrastructure,
 - puppetizing 143, 144
- Puppet Labs OpenStack module,
 - using 141-143

OpenStack modules

- URL 129

operating system 168

order management

- about 25, 26
- run stages 27

P

package management 280

Packer

- URL 274

PagerDuty service

- URL 250

parameters

- application 96
- component 96
- country 96
- datacenter 96
- defining, for each node 95-97
- entry points 123
- env 95
- file-based configurations 123
- naming standards 124
- project 96
- role 95
- setting-based configurations 123
- site 96
- tenant 96
- using 122, 123
- zone 96

params pattern 117, 118

Passenger

- about 204
- configuring 205-208
- installing 205-208

Pathogen

- installing 181
- URL 181

performance dashboard 74, 75

performance dashboard, metrics

- Catalog duplication 75
- Command processing 75
- Command queue 75
- Discarded 76
- JVM Heap memory usage 75
- Nodes 75
- Processed 75
- Rejected 76

- Resources 75
- Retried 75
- performance measurement**
 - about 221
 - code optimization 223, 224
 - Puppet Metrics 222
 - Puppet versions, testing 225
- pluginsync**
 - about 231, 232
 - for Ruby facts distribution 240-242
- PostgreSQL**
 - reference link 70
- postgresql module**
 - URL 244
- profiles**
 - about 91, 101, 102
 - managing 102, 103
- proxy mode, Puppet**
 - SNMP 258
 - Telnet or SSH connections 258
 - Web API 258
- proxy mode, with Puppet device application**
 - about 259, 260
 - reference 259
- Public modules 93**
- Puppet**
 - and Docker 271-274
 - configuring 5-7
 - executing 8, 9
 - installing 5-7
 - native mode 258
 - on storage devices 271
 - proxy mode 258
 - scaling 202
 - URL, for installation 6
- Puppet 3 automatic parameter lookup 51, 52**
- Puppet 4.x**
 - future enhancement 281
- Puppet 5 276**
- Puppet architecture**
 - components 92, 93
 - tasks 92
- Puppet architecture, examples**
 - about 104
 - default approach 105-107
 - ENC, used with Hieradata backend 108
 - facts driven approach 112, 113
 - Foreman and Hieradata, using 109
 - Foreman smart variables, using 111
 - Hieradata-based setup 110
 - nodeless site.pp 113, 114
- Puppetboard**
 - about 76
 - URL 76
- Puppet changes**
 - propagating 197
- Puppet code**
 - deploying 194
 - deploying, with librarian-puppet 195
 - deploying, with r10k 196
 - modifications 89, 90
 - testing 187
 - testing, with Beaker 192-194
 - testing, with beaker-rspec 192-194
 - testing, with rspec-puppet 188-190
 - testing, with Vagrant 190, 191
 - writing 179, 180
 - writing, in Geppetto 180
 - writing, in Vim 181
- Puppet code, stages**
 - configuration 228
 - instantiation 228
 - parsing and compiling 228
 - report 228
 - transport 228
- Puppet Collections 5**
- Puppet Conf**
 - reference link 282
- puppet.conf file, parameters**
 - ca_ttl 212
 - dns_alt_names 212
- PuppetDB**
 - about 3
 - configuration 64-67
 - installation 64-67
 - URL, for installation 65
- PuppetDB API**
 - about 76
 - commands, using 79, 80
 - queries, using 76-79
- puppetdb module**
 - reference link 73

- puppetdbquery module**
 - about 86
 - Hiera backend 88, 89
 - query format 87
 - querying, from command line 87
 - querying, from Puppet manifests 88
- puppetdb-termini package**
 - installing 73
- Puppet DSL**
 - user variables 20, 21
- Puppet ecosystem**
 - about 2, 3
 - configuration management, importance 3, 4
- Puppet Enterprise 2, 3**
- Puppet extension**
 - about 230
 - ENC, extendibility 230
 - Hiera, extendibility 230
 - indirector 232-236
 - pluginsync 231, 232
- Puppet, for cloud and virtualization**
 - about 264
 - Amazon Web Services 267
 - AWS provisioning 269
 - Cloud provisioning 270
 - VMware 265
- Puppet Forge 138**
- Puppet-friendly software, characteristics**
 - client-side registration 176
 - composable configuration files, using 175
 - disabled services, installing 176
 - online configuration reload 176
 - standard packaging system, using 175
- Puppet Labs OpenStack module**
 - URL 141
 - using 141, 143
- puppet-lint**
 - about 181
 - URL 187
- Puppet manifests**
 - puppetdbquery module, querying 88
- Puppet Master**
 - based on Trapperkeeper 209, 210
 - PuppetDB, configuring 73, 74
 - with Passenger 204
- Puppet Metrics 222**
- puppet-module-data**
 - reference link 119
- Puppet Modules Forge 115**
- Puppet, on network equipment**
 - about 257-259
 - cultural challenge 258
 - native Puppet, on network equipment 261
 - proxy mode, with puppet device
 - application 259-261
 - technical challenge 258
- Puppet OpenStack modules**
 - URL 139
- Puppet run**
 - anatomy 228-230
- Puppet run times**
 - distributing 218, 219
- puppetserver.conf file 7**
- Puppet versions**
 - testing 225

Q

- queries**
 - endpoints 77
 - using 77-79
- query parameters 77**

R

- r10k**
 - Puppet code, deploying with 196
- rake 188**
- RBCommons**
 - URL 187
- RedHat 222**
- reserved names**
 - about 28
 - reference link 28
- Resource Abstraction Layer (RAL) 10, 11, 228**
- resource declaration**
 - attributes 9
 - title 9
 - type 9
- resource defaults 17**
- resource references 17**
- resources 9, 10**

REST endpoints

- /aggregated-event-counts endpoint 86
- /catalogs endpoint 83
- /event-counts endpoint 85
- /events endpoint 84
- /fact-names endpoint 83
- /facts endpoint 81
- /metrics endpoint 83
- /nodes endpoint 82
- /reports endpoint 84
- /resources endpoint 81
- /server-time endpoint 86
- /version endpoint 86

reusability patterns

- about 125
- files, managing 126-128

reusable modules

- characteristics 125

roles 91, 101-103

roles and profiles pattern 137

rspec-puppet

- about 181
- URL 188
- using 188-190

Ruby 2

Ruby facts 240, 242

Ruby version

- URL 166

Ruby Version Manager (RVM)

- URL 198

run stages 27

S

scopes 18

selectors 28, 117

serialization format

- modifying 276, 277

settings-based OpenStack's

- official modules 143

shared modules

- about 97
- configuration files, placing 98

site module(s)

- about 95, 98
- configuration files, placing 98

smart variables 111

SSL termination

- managing 213-215

stack modules

- reusability 144-149

stage resource type 27

standard de facto pattern

- drawbacks 120
- pros and cons 120

standardize naming conventions 124

states 25

static files

- using 126

static scoping 24

stdlib module 121

stdmod

- URL 125

stdmod shared module

- URL 129

stored configs

- scaling 219-221

subqueries

- building 77

system integrations 274

system's facts 18

System Under Test (SUT) 192

T

tasks 91

templates

- using 126

termini 232

Tiny Puppet

- about 149, 150
- tp::conf 151
- tp::dir 151
- tp::install 151
- tp::repo 151
- URL 149

top scope variables 96

traffic compression 217

Trapperkeeper 209, 210

Travis

- about 198, 199
- URL 198

type-checking 21, 22

U

user-defined variables 4

users

managing 132, 133

user variables

in ENC 22

in Hierarchical 22

in Puppet DSL 20, 21

V

Vagrant

URL 190

using 190, 191

vagrant commands

vagrant destroy [machine] 191

vagrant halt [machine] 191

vagrant provision [machine] 191

vagrant ssh <machine> 191

vagrant status 191

vagrant up [machine] 191

variables 18, 109

variables scope

about 24, 25

class scope 24

node scope 24

sub class scope 24

top scope 24

vCenter 265

Vim 181

Vim bundles

Syntastic 181

vim-puppet 181

vim-puppet

about 181

URL 181

Virtual Machines (VM) 190

virtual resources 32

VMware

about 265

vCenter configuration 265

VM provisioning, on vCenter
and vSphere 265

vSphere virtual machines management,
with resource types 266, 267

W

Webrick 204

webserver.conf file 7

X

X-Client-DN 214

X-Client-Verify 214

X-SSL-Subject 214

