

**Felix Frank, Martin Alfke,
Alessandro Franceschi,
Jaime Soriano Pastor, Thomas Uphillis**

Puppet Mastering Infrastructure Automation

Learning Path

Gain key skills to manage your IT infrastructure and succeed with everyday IT automation



Packt

Puppet: Mastering Infrastructure Automation

**Gain key skills to manage your IT infrastructure and
succeed with everyday IT automation**

A course in three modules

Packt>

BIRMINGHAM - MUMBAI

Puppet: Mastering Infrastructure Automation

Copyright © 2017 Packt Publishing

All rights reserved. No part of this course 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 course to ensure the accuracy of the information presented. However, the information contained in this course 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 course.

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

Published on: June 2017

Production reference: 1100617

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

ISBN: 978-1-78839-970-8

www.packtpub.com

Credits

Authors

Felix Frank

Martin Alfke

Alessandro Franceschi

Jaime Soriano Pastor

Thomas Uphill

Reviewers

Ger Apeldoorn

Jeroen Hooyberghs

Bas Grolleman

Content Development Editor

Devika Battike

Graphics

Kirk D'penha

Production Coordinator

Aparna Bhagat

Preface

Puppet is a configuration management tool that allows you to automate all your IT configurations, giving you control. It was written for and by system administrators to manage large numbers of systems efficiently and prevent configuration drifts..

What this learning path covers

Module 1, Puppet 4 Essentials, Second Edition, gets you started rapidly and intuitively as you'll put Puppet's tools to work right away. It will also highlight the changes associated with performance improvements as well as the new language features in Puppet 4. We'll start with a quick introduction to Puppet to get you managing your IT systems quickly. You will then learn about the Puppet Agent that comes with an all-in-one (AIO) package and can run on multiple systems. Next, we'll show you the Puppet Server for high-performance communication and passenger packages. As you progress through the book, the innovative structure and approach of Puppet will be explained with powerful use cases. The difficulties that are inherent to a complex and powerful tool will no longer be a problem for you as you discover Puppet's fascinating intricacies.

By the end of this module, you will not only know how to use Puppet, but also its companion tools Facter and Hiera, and will be able to leverage the flexibility and expressive power implemented by their tool chain.

Module 2, Extending Puppet - Second Edition , will be your guide to designing and deploying your Puppet architecture. It will help you utilize Puppet to manage your IT infrastructure. Get to grips with Hiera and learn how to install and configure it, before learning best practices for writing reusable and maintainable code. You will also be able to explore the latest features of Puppet 4, before executing, testing, and deploying Puppet across your systems. As you progress, Extending Puppet takes you through higher abstraction modules, along with tips for effective code workflow management.

Finally, you will learn how to develop plugins for Puppet - as well as some useful techniques that can help you to avoid common errors and overcome everyday challenges.

Module 3, Mastering Puppet Second Edition, deals with the issues faced when scaling out Puppet to handle large numbers of nodes. It will show you how to fit Puppet into your enterprise and allow many developers to work on your Puppet code simultaneously. In addition, you will learn to write custom facts and roll your own modules to solve problems. Next, popular options for performing reporting and orchestration tasks will be introduced in this book. Moving over to troubleshooting techniques, which will be very useful. The concepts presented are useful to any size organization.

By the end of this module, you will know how to deal with problems of scale and exceptions in your code, automate workflows, and support multiple developers working simultaneously.

What you need for this learning path

The primary requirements are as follows:

- A Debian GNU/Linux operating system in version 7, code name "Wheezy"
- A Linux system connected to the internet.
- Additional repositories used were EPEL (Extra Packages for Enterprise Linux), the Software Collections (SCL) repository, the Foreman repository, and the Puppet Labs
- An Enterprise Linux 7 derived installation, such as CentOS 7, Scientific Linux 7, or Springdale Linux 7.
- The Latest version of Puppet

Who this learning path is for

If you're an experienced IT professional and a new Puppet user, this course will provide you with all you need to know to go from installation to advanced automation. Get a rapid introduction to the essential topics and then tackle Puppet for advanced automation. Through a practical approach and innovative selection of topics, you'll explore how to design, implement, adapt, and deploy a Puppet architecture.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – 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 course'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 course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

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

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at :

<https://github.com/PacktPublishing/Puppet-Mastering-Infrastructure-Automation>

We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

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 courses – 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 course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, 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 course 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 course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1: Puppet 4 Essentials Second Edition

Chapter 1: Writing Your First Manifests	3
Getting started	4
Introducing resources and properties	6
Interpreting the output of the puppet apply command	7
Adding control structures in manifests	9
Using variables	11
Controlling the order of evaluation	12
Implementing resource interaction	20
Examining the most notable resource types	22
Summary	28
Chapter 2: The Master and Its Agents	29
The Puppet Master	29
Setting up the Puppet Agent	34
Performance considerations	42
Using Phusion Passenger with Nginx	43
Completing the stack with PuppetDB	46
Troubleshooting SSL issues	46
Summary	48
Chapter 3: A Peek under the Hood – Facts, Types, and Providers	49
Summarizing systems with Facter	49
Understanding the type system	58
Substantiating the model with providers	61
Putting it all together	63
Summary	65

Table of Contents

Chapter 4: Modularizing Manifests with Classes and Defined Types	67
Introducing classes and defined types	68
Structured design patterns	73
Including classes from defined types	84
Establishing relationships among containers	84
Making classes more flexible through parameters	92
Summary	95
Chapter 5: Extending Your Puppet Infrastructure with Modules	97
An overview of Puppet's modules	98
Maintaining environments	101
Building a specific module	108
Finding helpful Forge modules	132
Summary	133
Chapter 6: Leveraging the Full Toolset of the Language	135
Templating dynamic configuration files	136
Creating virtual resources	139
Exporting resources to other agents	144
Overriding resource parameters	151
Avoiding antipatterns	155
Summary	157
Chapter 7: New Features from Puppet 4	159
Upgrading to Puppet 4	160
Using the type system	164
Learning lambdas and functions	172
Creating Puppet 4 functions	176
Leveraging the new template engine	179
Handling multiline with HEREDOC	180
Breaking old practices	183
Summary	189
Chapter 8: Separating Data from Code Using Hiera	191
Understanding the need for separate data storage	192
Structuring configuration data in a hierarchy	195
Retrieving and using Hiera values in manifests	200
Converting resources to data	207
Debugging Hiera lookups	214
Implementing the Roles and Profiles pattern	214
Summary	216

Module 2: Extending Puppet, Second Edition

Chapter 1: Puppet Essentials	221
The Puppet ecosystem	222
Puppet components	224
Installing and configuring Puppet	225
Puppet in action	228
Variables, facts, and scopes	238
Meta parameters	245
Managing order and dependencies	245
Reserved names and allowed characters	248
Conditionals	248
Comparison operators	249
Iteration and lambdas	250
Exported resources	251
Modules	253
Restoring files from a filebucket	256
Summary	258
Chapter 2: Managing Puppet Data with Hiera	259
Installing and configuring Hiera	260
Working with the command line on a YAML backend	264
Using Hiera in Puppet	269
Additional Hiera backends	273
Using Hiera as an ENC	280
Summary	281
Chapter 3: Introducing PuppetDB	283
Installation and configuration	284
Dashboards	294
PuppetDB API	296
Querying PuppetDB for fun and profit	300
The puppetdbquery module	306
How Puppet code may change in the future	309
Summary	310
Chapter 4: Designing Puppet Architectures	311
Components of a Puppet architecture	312
The Foreman	319
Roles and profiles	321

Table of Contents —————

The data and the code	323
Sample architectures	324
Summary	334
Chapter 5: Using and Writing Reusable Modules	335
Modules layout evolution	336
The parameters dilemma	342
Reusability patterns	345
Summary	356
Chapter 6: Higher Abstraction Modules	357
The OpenStack example	359
An approach to reusable stack modules	364
Tiny Puppet	369
Summary	371
Chapter 7: Puppet Migration Patterns	373
Examining potential scenarios and approaches	373
Patterns for extending Puppet coverage	380
Things change	391
Summary	397
Chapter 8: Code Workflow Management	399
Write Puppet code	399
Git workflows	402
Code review	406
Testing Puppet code	407
Deploying Puppet code	414
Propagating Puppet changes	416
Puppet continuous integration	417
Summary	420
Chapter 9: Scaling Puppet Infrastructures	421
Scaling Puppet	422
Measuring performance	441
Summary	446
Chapter 10: Extending Puppet	447
Anatomy of a Puppet run, under the hood	448
Puppet extension alternatives	450
Custom functions	456
Custom facts	459
Custom types and providers	463

Table of Contents

Custom report handlers	470
Custom faces	472
Summary	476
Chapter 11: Beyond the System	477
Puppet on network equipment	477
Puppet for cloud and virtualization	484
Puppet on storage devices	491
Puppet and Docker	491
Summary	494
Chapter 12: Future Puppet	495
Changing the serialization format	496
Direct Puppet	498
Other changes	499
Beyond Puppet 4.x	501
Summary	502

Module 3: Mastering Puppet, Second Edition

Chapter 1: Dealing with Load/Scale	507
Divide and conquer	507
Conquer by dividing	527
Summary	533
Chapter 2: Organizing Your Nodes and Data	535
Getting started	535
Organizing the nodes with an ENC	535
Hiera	550
Summary	560
Chapter 3: Git and Environments	561
Environments	561
Git	570
Git for everyone	595
Summary	598
Chapter 4: Public Modules	599
Getting modules	599
Using GitHub for public modules	599
Modules from the Forge	602
Using Librarian	604
Using r10k	606

Table of Contents

Using Puppet-supported modules	612
Summary	632
Chapter 5: Custom Facts and Modules	633
Module manifest files	634
Custom facts	647
CFacter	656
Summary	657
Chapter 6: Custom Types	659
Parameterized classes	659
Defined types	661
Types and providers	672
Summary	680
Chapter 7: Reporting and Orchestration	681
Turning on reporting	681
Store	682
Logback	683
Internet relay chat	683
Foreman	688
Puppet GUIs	693
mcollective	693
Ansible	705
Summary	705
Chapter 8: Exported Resources	707
Configuring PuppetDB – using the Forge module	707
Manually installing PuppetDB	711
Exported resource concepts	715
Resource tags	718
Exported SSH keys	719
Putting it all together	723
Summary	732
Chapter 9: Roles and Profiles	733
Design pattern	733
Creating an example CDN role	734
Dealing with exceptions	740
Summary	741

Table of Contents

<u>Chapter 10: Troubleshooting</u>	743
Connectivity issues	744
Catalog failures	747
Debugging	752
Summary	756
<u>Bibliography</u>	757
<u>Index</u>	759

Module 1

Puppet 4 Essentials, Second Edition

Acquire the skills to manage your IT infrastructure effectively with Puppet

1

Writing Your First Manifests

Over the last few years, configuration management has become increasingly significant to the IT world. Server operations in particular is hardly even feasible without a robust management infrastructure. Among the available tools, Puppet has established itself as one of the most popular and widespread solutions. Originally written by *Luke Kanies*, the tool is now distributed under the terms of Apache License 2.0 and maintained by Luke's company, Puppet Labs. It boasts a large and bustling community, rich APIs for plugins and supporting tools, outstanding online documentation, and a great security model based on SSL authentication.

Like all configuration management systems, Puppet allows you to maintain a central repository of infrastructure definitions, along with a toolchain to enforce the desired state on the systems under management. The whole feature set is quite impressive. This book will guide you through some steps to quickly grasp the most important aspects and principles of Puppet.

In this chapter, we will cover the following topics:

- Getting started
- Introducing resources and properties
- Interpreting the output of the `puppet apply` command
- Adding control structures in manifests
- Using variables
- Controlling the order of evaluation
- Implementing resource interaction
- Examining the most notable resource types

Getting started

Installing Puppet is easy. On large Linux distributions, you can just install the Puppet package via apt-get or yum.

The installation of Puppet can be done in the following ways:

- From default Operating System repositories
- From Puppet Labs

The former way is generally simpler. *Chapter 2, The Master and Its Agents*, provides simple instructions to install the Puppet Labs packages. A platform-independent way to install Puppet is to get the puppet Ruby gem. This is fine for testing and managing single systems, but is not recommended for production use.

After installing Puppet, you can use it to do something for you right away. Puppet is driven by manifests, the equivalent of scripts or programs, written in Puppet's **domain-specific language (DSL)**. Let's start with the obligatory *Hello, world!* manifest:

```
# hello_world.pp
notify { 'Hello, world!':
}
```

Downloading the example code



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

To put the manifest to work, use the following command. (We avoided the term "execute" on purpose—manifests cannot be executed. More details will follow around the middle of this chapter.):

```
root@puppetmaster:~# puppet apply hello_world.pp
Notice: Compiled catalog for puppetmaster.example.net in environment
production in 0.45 seconds
Notice: Hello, world!
Notice: /Stage[main]/Main/Notify[Hello, world!]/message: defined
'message' as 'Hello, world!'
Notice: Applied catalog in 0.03 seconds
```

Before we take a look at the structure of the manifest and the output from the puppet apply command, let's do something useful, just as an example. Puppet comes with its own background service. Let's assume that you want to learn the basics before letting it mess with your system. You can write a manifest to have Puppet make sure that the service is not currently running and will not be started at system boot:

```
# puppet_service.pp
service { 'puppet':
  ensure => 'stopped',
  enable => false,
}
```

To control system processes, boot options, and the like, Puppet needs to be run with root privileges. This is the most common way to invoke the tool, because Puppet will often manage OS-level facilities. Apply your new manifest with root access, either through sudo, or from a root shell, as shown in the following transcript:

```
root@puppetmaster:~# puppet apply puppet_service.pp
Notice: Compiled catalog for puppetmaster.example.net in environment
production in 0.61 seconds
Notice: /Stage[main]/Main/Service[puppet]/ensure: ensure changed
'running' to 'stopped'
Notice: Applied catalog in 0.15 seconds
```

Now, Puppet has disabled the automatic startup of its background service for you. Applying the same manifest again has no effect, because the necessary steps are already complete:

```
root@puppetmaster:~# puppet apply puppet_service.pp
Notice: Compiled catalog for puppetmaster.example.net in environment
production in 0.62 seconds
Notice: Applied catalog in 0.07 seconds
```

This reflects a standard behavior in Puppet: Puppet resources are **idempotent**—which means that every resource first compares the actual (system) with the desired (puppet) state and only initiates actions in case there is a difference (configuration drift).

You will often get this output, as shown previously, from Puppet. It tells you that everything is as it should be. As such, this is a desirable outcome, like the all clean output from git status.

Introducing resources and properties

Each of the manifests you wrote in the previous section declared one respective resource. Resources are the elementary building blocks of manifests. Each has a type (in this case, `notify` and `service`, respectively) and a name or title (`Hello`, `world!` and `puppet`). Each resource is unique to a manifest, and can be referenced by the combination of its type and name, such as `Service["puppet"]`. Finally, a resource also comprises a list of zero or more attributes. An attribute is a key-value pair, such as `"enable => false"`.

Attribute names cannot be chosen arbitrarily. Each resource type supports a specific set of attributes. Certain parameters are available for all resource types (metaparameters), and some names are just very common, such as `ensure`. The `service` type supports the `ensure` property, which represents the status of the managed process. Its `enabled` property, on the other hand, relates to the system boot configuration (with respect to the service in question).

Note that we have used the terms attribute, property, and parameter in a seemingly interchangeable fashion. Don't be deceived – there are important distinctions. Property and parameter are the two different kinds of attributes that Puppet uses. You have already seen two properties in action. Let's look at a parameter:

```
service { 'puppet':  
  ensure  => 'stopped',  
  enable   => false,  
  provider => 'upstart',  
}
```

The `provider` parameter tells Puppet that it needs to interact with the `upstart` subsystem to control its background service, as opposed to `systemd` or `init`. If you don't specify this parameter, Puppet makes a well-educated guess. There is quite a multitude of supported facilities to manage services on a system. You will learn more about providers and their automatic choosing later on.

The difference between parameters and properties is that the parameter merely indicates *how* Puppet should manage the resource, not what a desired state is. Puppet will only take action on property values. In this example, these are `ensure => 'stopped'` and `enable => false`. For each such property, Puppet will perform the following tasks:

- Test whether the resource is already in sync with the target state
- If the resource is not in sync, it will trigger a sync action

A property is considered to be in sync when the system entity that is managed by the given resource (in this case, the upstart service configuration for Puppet) is in the state that is described by the property value in the manifest. In this example, the ensure property will be in sync only if the puppet service is not running. The enable property is in sync if upstart is not configured to launch Puppet at system start.

As a mnemonic concerning parameters versus properties, just remember that properties can be out of sync, whereas parameters cannot.

Puppet also allows you to read your existing system state by using the puppet resource command:

```
root@puppetmaster:~# puppet resource user root
user { 'root':
  ensure          => 'present',
  comment         => 'root',
  gid             => '0',
  home            => '/root',
  password        => '$6$17/7FtU/$TvYEDtFgGr0SaS7xOVloWXVTqQxxDUGH.
eBKJ7bgHJ.hdoc03Xrvvm2ru0HFKpu1QSpVW/7o.rLdk/9MZANEgt/',
  password_max_age => '99999',
  password_min_age => '0',
  shell            => '/bin/bash',
  uid              => '0',
}
}
```

Please note that some resource types will return read-only attributes (for example, the file resource type will return `mtime` and `ctime`). Refer to the appropriate type documentation.

Interpreting the output of the puppet apply command

As you have already witnessed, the output presented by Puppet is rather verbose. As you get more experienced with the tool, you will quickly learn to spot the crucial pieces of information. Let's first take a look at the informational messages though. Apply the `service.pp` manifest once more:

```
root@puppetmaster:~# puppet apply puppet_service.pp
```

```
Notice: Compiled catalog for puppetmaster.example.net in environment  
production in 0.48 seconds
```

```
Notice: Applied catalog in 0.05 seconds
```

Puppet took no particular action. You only get two timings: one, from the compiling phase of the manifest, and the other, from the catalog application phase. The catalog is a comprehensive representation of a compiled manifest. Puppet bases all its efforts concerning the evaluation and syncing of resources on the content of its current catalog.

Now, to quickly force Puppet to show you some more interesting output, pass it a one-line manifest directly from the shell. Regular users of Ruby or Perl will recognize the call syntax:

```
# puppet apply -e'service { "puppet": enable => true, }'  
Notice: Compiled catalog for puppetmaster.example.net in environment  
production in 0.62 seconds  
Notice: /Stage[main]/Main/Service[puppet]/enable: enable changed 'false'  
to 'true'  
Notice: Applied catalog in 0.12 seconds.
```



We prefer double quotes in manifests that get passed as command-line arguments, because on the shell, the manifest should be enclosed in single quotes as a whole.



You instructed Puppet to perform yet another change on the Puppet service. The output reflects the exact change that was performed. Let's analyze this log message:

- The `Notice:` keyword at the beginning of the line represents the log level. Other levels include `Warning`, `Error`, and `Debug`.
- The property that changed is referenced with a whole path, starting with `Stage [main]`. Stages are beyond the scope of this book, so you will always just see the default of `main` here.
- The next path element is `Main`, which is another default. It denotes the class in which the resource was declared. You will learn about classes in *Chapter 4, Modularizing Manifests with Classes and Defined Types*.
- Next, is the resource. You already learned that `Service [puppet]` is its unique reference.
- Finally, `enable` is the name of the property in question. When several properties are out of sync, there will usually be one line of output for each property that gets synchronized.

- The rest of the log line indicates the type of change that Puppet saw fit to apply. The wording depends on the nature of the property. It can be as simple as `created`, for a resource that is newly added to the managed system, or a short phrase such as `changed false to true`.

Dry-testing your manifest

Another useful command-line switch for `puppet apply` is the `--noop` option. It instructs Puppet to refrain from taking any action on unsynced resources. Instead, you only get a log output that indicates what will change without the switch. This is useful in determining whether a manifest would possibly break anything on your system:

```
root@puppetmaster:~# puppet apply puppet_service.pp --noop
Notice: Compiled catalog for puppetmaster.example.net in environment
production in 0.63 seconds
Notice: /Stage[main]/Main/Service[puppet]/enable: current_value true,
should be false (noop)
Notice: Class[Main]: Would have triggered 'refresh' from 1 events
Notice: Stage[main]: Would have triggered 'refresh' from 1 events
Notice: Applied catalog in 0.06 seconds
```

Note that the output format is the same as before, with a `(noop)` marker trailing the notice about the sync action. This log can be considered a preview of what will happen when the manifest is applied without the `--noop` switch.

The additional notices about triggered refreshes will be described later and can be ignored for the moment. You will have a better understanding of their significance after finishing this chapter and *Chapter 4, Modularizing Manifests with Classes and Defined Types*.

Adding control structures in manifests

You have written three simple manifests while following the instructions in this chapter so far. Each comprised only one resource, and one of them was given on the command line using the `-e` option. Of course, you would not want to write distinct manifests for each possible circumstance. Instead, just as how Ruby or Perl scripts branch out into different code paths, there are structures that make your Puppet code flexible and reusable for different circumstances.

The most common control element is the `if/else` block. It is quite similar to its equivalents in many programming languages:

```
if 'mail_lda' in $needed_services {  
    service { 'dovecot': enable => true }  
} else {  
    service { 'dovecot': enable => false }  
}
```

The Puppet DSL also has a `case` statement, which is reminiscent of its counterparts in other languages as well:

```
case $role {  
    'imap_server': {  
        package { 'dovecot': ensure => 'installed' }  
        service { 'dovecot': ensure => 'running' }  
    }  
    /_webserver$/ : {  
        service { [ 'apache', 'ssh' ]: ensure => 'running' }  
    }  
    default: {  
        service { 'ssh': ensure => running }  
    }  
}
```

A variation of the `case` statement is the selector. It's an expression, not a statement, and can be used in a fashion similar to the ternary `if/else` operator found in C-like languages:

```
package { 'dovecot':  
    ensure => $role ? {  
        'imap_server' => 'installed',  
        /desktop$/     => 'purged',  
        default         => 'removed',  
    },  
}
```

It should be used with caution, because in more complex manifests, this syntax will impede readability.

Using variables

Variable assignment works just like in most scripting languages. Any variable name is always prefixed with the \$ sign:

```
$download_server = 'img2.example.net'  
$url = "https://{$download_server}/pkg/example_source.tar.gz"
```

Also, just like most scripting languages, Puppet performs variable value substitution in strings that are in double quotes, but no interpolation at all in single-quoted strings.

Variables are useful for making your manifest more concise and comprehensible. They help you with the overall goal of keeping your source code free from redundancy. An important distinction from variables in imperative programming and scripting languages is the **immutability** of variables in Puppet manifests. Once a value has been assigned, it cannot be overwritten.

Why is it called a variable at all if it is a constant? One should never look at Puppet as a tool that manages a single system. For a single system, a Puppet variable might look like a constant. But Puppet manages a multitude of systems with different operating systems. Across all these systems, variables will be different and not constants.

Variable types

As of Puppet 3.x, there are only four variable types: Strings, Arrays, Hashes, and Boolean. Puppet 4 introduces a rich type system. The new type system will be explained in *Chapter 7, New Features from Puppet 4*. The basic variable types work much like their respective counterparts in other languages. Depending on your background, you might be familiar with using associative arrays or dictionaries as semantic equivalents to Puppet's hash type:

```
$a_bool = true  
$a_string = 'This is a string value'  
$an_array = [ 'This', 'forms', 'an', 'array' ]  
$a_hash = {  
    'subject' => 'Hashes',  
    'predicate' => 'are written',  
    'object' => 'like this',  
    'note' => 'not actual grammar!',  
    'also note' => [ 'nesting is',  
        { 'allowed' => 'of course' } ],  
}
```

Accessing the values is equally simple. Note that the hash syntax is similar to that of Ruby, not Perl's:

```
$x = $a_string  
$y = $an_array[1]  
$z = $a_hash['object']
```

Strings can be used as resource attribute values, but it's worth noting that a resource title can also be a variable reference:

```
package { $apache_package:  
    ensure => 'installed'  
}
```

It's intuitively clear what a string value means in this context. But you can also pass arrays here to declare a whole set of resources in one statement. The following manifest manages three packages, making sure that they are all installed:

```
$packages = [ 'apache2',  
    'libapache2-mod-php5',  
    'libapache2-mod-passenger', ]  
package { $packages:  
    ensure => 'installed'  
}
```

You will learn how to make efficient use of hash values in later chapters.

The array does not *need* to be stored in a variable to be used, but it is a good practice in some cases.

Controlling the order of evaluation

With what you've seen this far, you might have got the impression that Puppet's DSL is a specialized scripting language. That is actually quite far from the truth—a manifest is not a script or program. The language is a tool to model a system state through a set of resources, including files, packages, and cron jobs, among others.

The whole paradigm is different from that of scripting languages. Ruby or Perl are imperative languages that are based around statements that will be evaluated in a strict order. The Puppet DSL is declarative, which means that the manifest declares a set of resources that are expected to have certain properties. These resources are put into a catalog, and Puppet then tries to build a path through all declared resources. The compiler parses the manifests in order, but the configurer applies resources in a very different way.

In other words, the manifests should always describe what you expect to be the end result. The specifics of what actions need to be taken to get there are decided by Puppet.

To make this distinction more clear, let's look at an example:

```
package { 'haproxy':
  ensure => 'installed',
}
file {'/etc/haproxy/haproxy.cfg':
  ensure => file,
  owner  => 'root',
  group  => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
}
service { 'haproxy':
  ensure => 'running',
}
```

With this manifest, Puppet will make sure that the following state is reached:

1. The HAProxy package is installed.
2. The `haproxy.cfg` file has specific content, which has been prepared in a file in `/etc/puppet/modules/`.
3. HAProxy is started.

To make this work, it is important that the necessary steps are performed in order.

1. A configuration file cannot usually be installed before the package, because there is not yet a directory to contain it.
2. The service cannot start before installation either. If it becomes active before the configuration is in place, it will use the default settings from the package instead.

This point is being stressed because the preceding manifest does not, in fact, contain cues for Puppet to indicate such a strict ordering. Without explicit dependencies, Puppet is free to put the resources in any order it sees fit.

The recent versions of Puppet allow a form of local manifest-based ordering, so the presented example will actually work as is. The manifest-based ordering can be configured in the `puppet.conf` configuration file as follows:



```
ordering = manifest.
```

This setting is default for Puppet 4. It is still important to be aware of the ordering principles, because the implicit order is difficult to determine in more complex manifests, and as you will learn soon, there are other factors that will influence the order.

Declaring dependencies

The easiest way to bring order to such a straightforward manifest is resource chaining. The syntax for this is a simple ASCII arrow between two resources:

```
package { 'haproxy':
  ensure => 'installed',
}
->
file { '/etc/haproxy/haproxy.cfg':
  ensure => file,
  owner  => 'root',
  group  => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
}
->
service {'haproxy':
  ensure => 'running',
}
```

This is only viable if all the related resources can be written next to each other. In other words, if the graphic representation of the dependencies does not form a straight chain, but more of a tree, star, or any other shape, this syntax is not sufficient.



Internally, Puppet *will* construct an ordered graph of resources and synchronize them during a traversal of that graph.

A more generic and flexible way to declare dependencies is through special **metaparameters** – parameters that are eligible for use with any resource type. There are different metaparameters, most of which have nothing to do with ordering (you have seen `provider` in an earlier example).

For resource ordering, puppet offers the metaparameters, require and before. Both take one or more references to a declared resource as their value. Puppet references have a special syntax, as was previously mentioned:

```
Type['title']
e.g.
Package['haproxy']
```



Please note that you can only build references to resources which are declared in the catalog. You cannot build and use references to something that is not managed by Puppet, even when it exists on the managed system.

Here is the HAproxy manifest, ordered using the require metaparameter:

```
package { 'haproxy':
  ensure => 'installed',
}
file {'/etc/haproxy/haproxy.cfg':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
  require => Package['haproxy'],
}
service {'haproxy':
  ensure => 'running',
  require => File['/etc/haproxy/haproxy.cfg'],
}
```

The following manifest is semantically identical, but relies on the before metaparameter rather than require:

```
package { 'haproxy':
  ensure => 'installed',
  before => File['/etc/haproxy/haproxy.cfg'],
}
file { '/etc/haproxy/haproxy.cfg':
  ensure => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
  before  => Service['haproxy'],
```

```
    }
    service { 'haproxy':
      ensure => 'running',
    }
```



The manifest can also mix both styles of notation, of course.
This is left as a reader exercise with no dedicated depiction.



The `require` metaparameter usually leads to more understandable code, because it expresses the dependency of the annotated resource on another resource. The `before` parameter, on the other hand, implies a dependency that a referenced resource forms upon the current resource. This can be counter-intuitive, especially for frequent users of packaging systems (which usually implement a `require` style dependency declaration).

Sometimes, it might be difficult to decide whether to use `require` or `before`. In simple cases, most people prefer `require`. In some cases, it is easier to use `before`. Think of services that have multiple configuration files. Keeping information about the configuration file and the requirement in a single place reduces errors caused by forgetting to also adopt changes to the service, when adding or removing additional configuration files. Take a look at the following example code:

```
file { '/etc/apache2/apache2.conf':
  ensure => file,
  before => Service['apache2'],
}
file { '/etc/apache2/httpd.conf':
  ensure => file,
  before => Service['apache2'],
}
service { 'apache2':
  ensure  => running,
  enable   => true,
}
```

In the example, all dependencies are declared within the file resource declarations. If you use the `require` parameter instead, you will always need to touch at least two resources in case of changes:

```
file { '/etc/apache2/apache2.conf':
  ensure => file,
}
file { '/etc/apache2/httpd.conf':
  ensure => file,
```

```
    }
    service { 'apache2':
      ensure  => running,
      enable  => true,
      require => [
        File['/etc/apache2/apache2.conf'],
        File['/etc/apache2/httpd.conf'],
      ],
    }
```

Will you remember to update the service resource declaration whenever you add a new file to be managed by Puppet?

Consider another simpler example:

```
if $os_family == 'Debian' {
  file { '/etc/apt/preferences.d/example.net.prefs':
    content => '...',
    before   => Package['apache2'],
  }
}
package { 'apache2':
  ensure => 'installed',
}
```

The file in the `preferences.d` directory only makes sense for Debian-like systems. That's why the package cannot safely require it. If the manifest is applied on a different OS, such as CentOS, the `apt` preferences file will not appear in the catalog thanks to the `if` clause. If the package had it as a requirement regardless, the resulting catalog would be inconsistent, and Puppet would not apply it. Specifying `before` in the file resource is safe, and semantically equivalent.

The `before` metaparameter is outright necessary in situations like this one, and can make the manifest code more elegant and straightforward in other scenarios. Familiarity with both `before` and `require` is advisable.

Error propagation

Defining requirements serves another important purpose. References on declared resources will only be validated as successful references if the depended upon resource was finished successfully. This can be seen like a stop point inside Puppet DSL code, when a required resource is not synchronized successfully.

For example, a file resource will fail if the URL of the source file is broken:

```
file { '/etc/haproxy/haproxy.cfg':
  ensure => file,
  source => 'puppet:///modules/haproxy/etc/haproxy.cfg',
}
```

One path segment is missing here. Puppet will report that the file resource could not be synchronized:

```
root@puppetmaster:~# puppet apply typo.pp
Notice: Compiled catalog for puppetmaster.example.net in environment
production in 0.62 seconds
Error: /Stage[main]/Main/File[/etc/haproxy/haproxy.cfg]: Could not
evaluate: Could not retrieve information from environment production
source(s) puppet:///modules/haproxy/etc/haproxy.cfg
Notice: /Stage[main]/Main/Service[haproxy]: Dependency File[/etc/haproxy/
haproxy.cfg] has failures: true
Warning: /Stage[main]/Main/Service[haproxy]: Skipping because of failed
dependencies
Notice: Applied catalog in 0.06 seconds
```

In this example, the Error line describes the error caused by the broken URL. The error propagation is represented by the Notice and Warning lines below it.

Puppet failed to apply changes to the configuration file; it cannot compare the current state to the nonexistent source. As the service depends on the configuration file, Puppet will not even try to start it. This is for safety: if any dependencies cannot be put into the defined state, Puppet must assume that the system is not fit for application of the dependent resource.

This is another important reason to make consequent use of resource dependencies. Remember that both the chaining arrow and the before metaparameter imply error propagation as well.

Avoiding circular dependencies

Before you learn about another way in which resources can interrelate, there is an issue that you should be aware of: dependencies must not form circles. Let's visualize this in an example:

```
file { '/etc/haproxy':
  ensure => 'directory',
  owner  => 'root',
```

```

group  => 'root',
mode   => '0644',
}
file { '/etc/haproxy/haproxy.cfg':
  ensure => file',
  owner  => 'root',
  group   => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
}
service { 'haproxy':
  ensure  => 'running',
  require => File['/etc/haproxy/haproxy.cfg'],
  before   => File['/etc/haproxy'],
}

```

The dependency circle in this manifest is somewhat hidden (as will likely be the case for many such circles that you will encounter during regular use of Puppet). It is formed by the following relations:

- The `File['/etc/haproxy/haproxy.cfg']` auto-requires the parent directory, `File['/etc/haproxy']`. This is an implicit, built-in dependency.
- The parent directory, `File['/etc/haproxy']`, requires `Service['haproxy']` due to its `before` metaparameter.
- The `Service['haproxy']` service requires the `File['/etc/haproxy/haproxy.cfg']` config.

Implicit dependencies exist for the following resource combinations, among others:



- If a directory and a file inside the directory is declared, Puppet will first create the directory and then the file
- If a user and his primary group is declared, Puppet will first create the group and then the user
- If a file and the owner (user) is declared, Puppet will first create the user and then the file

Granted, the preceding example is contrived—it will not make sense to manage the service before the configuration directory. Nevertheless, even a manifest design that is apparently sound can result in circular dependencies. This is how Puppet will react to that:

```

root@puppetmaster:~# puppet apply circle.pp
Notice: Compiled catalog for puppetmaster.example.net in environment
production in 0.62 seconds

```

```
Error: Failed to apply catalog: Found 1 dependency cycle:  
(File[/etc/haproxy/haproxy.cfg] => Service[haproxy] => File[/etc/haproxy]  
=> File[/etc/haproxy/haproxy.cfg])  
Try the '--graph' option and opening the resulting '.dot' file in  
OmniGraffle or GraphViz
```

The output helps you locate the offending relation(s). For very wide dependency circles with lots of involved resources, the textual rendering is difficult to analyze. Therefore, Puppet also gives you the opportunity to get a graphical representation of the dependency graph through the --graph option.

If you do this, Puppet will include the full path to the newly created .dot file in its output. Its content looks similar to Puppet's output:

```
digraph Resource_Cycles {  
    label = "Resource Cycles"  
    "File[/etc/haproxy/haproxy.cfg]" ->"Service[haproxy]" ->"File[/etc/  
haproxy]" ->"File[/etc/haproxy/haproxy.cfg]"  
}
```

This is not helpful by itself, but it can be fed directly into tools such as dotty to produce an actual diagram.

To summarize, resource dependencies are helpful in keeping Puppet from acting upon resources in unexpected or uncontrolled situations. They are also useful in restricting the order of resource evaluation.

Implementing resource interaction

In addition to dependencies, resources can also enter a similar yet different mutual relation. Remember the pieces of output that we skipped earlier. They are as follows:

```
root@puppetmaster:~# puppet apply puppet_service.pp --noop  
Notice: Compiled catalog for puppetmaster.example.net in environment  
production in 0.62 seconds  
Notice: /Stage[main]/Main/Service[puppet]/ensure: current_value running,  
should be stopped (noop)  
Notice: Class[Main]: Would have triggered 'refresh' from 1 events  
Notice: Stage[main]: Would have triggered 'refresh' from 1 events  
Notice: Applied catalog in 0.05 seconds
```

Puppet mentions that **refreshes** would have been triggered for the reason of an **event**. Such events are emitted by resources whenever Puppet acts on the need for a sync action. Without explicit code to receive and react to events, they just get discarded.

The mechanism to set up such event receivers is named in an analogy of a generic publish/subscribe queue—resources get configured to react to events using the `subscribe` metaparameter. There is no `publish` keyword or parameter, since each and every resource is technically a publisher of events (messages). Instead, the counterpart of the `subscribe` metaparameter is called `notify`, and it explicitly directs generated events at referenced resources.

One of the most common practical uses of the event system is to reload service configurations. When a `service` resource consumes an event (usually from a change in a config file), Puppet invokes the appropriate action to make the service restart.



If you instruct Puppet to do this, it can result in brief service interruptions due to this restart operation. Note that if the new configuration causes an error, the service might fail to start and stay offline.



The following code example shows the relationships between the haproxy package, the corresponding haproxy configuration file, and the haproxy service:

```
file { '/etc/haproxy/haproxy.cfg':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  source  => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
  require  => Package['haproxy'],
}
service { 'haproxy':
  ensure  => 'running',
  subscribe => File['/etc/haproxy/haproxy.cfg'],
}
```

If the `notify` metaparameter is to be used instead, it must be specified for the resource that emits the event:

```
file { '/etc/haproxy/haproxy.cfg':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
```

```
source  => 'puppet:///modules/haproxy/etc/haproxy/haproxy.cfg',
require => Package['haproxy'],
notify  => Service['haproxy'],
}
service { 'haproxy':
  ensure  => 'running',
}
```

This will likely feel reminiscent of the `before` and `require` metaparameters, which offer symmetric ways of expressing an interrelation of a pair of resources just as well. This is not a coincidence—these metaparameters are closely related to each other:

- The resource that subscribes to another resource implicitly requires it
- The resource that notifies another is implicitly placed before the latter one in the dependency graph

In other words, `subscribe` is the same as `require`, except for the dependent resource receiving events from its peer. The same holds true for `notify` and `before`.

The chaining syntax is also available for signaling. To establish a signaling relation between neighboring resources, use an ASCII arrow with a tilde, `~>`, instead of the dash in `->`:

```
file { '/etc/haproxy/haproxy.cfg': ... }
~>
service { 'haproxy': ... }
```

The `service` resource type is one of the two notable types that support **refreshing** when resources get notified (the other will be discussed in the next section). There are others, but they are not as ubiquitous.

Examining the most notable resource types

To complete our tour of the basic elements of a manifest, let's take a closer look at the resource types that you have already used, and some of the more important ones that you have not yet encountered.

You probably already have a good feeling for the `file` type, which will ensure the existence of files and directories, along with their permissions. Pulling a file from a repository (usually, a Puppet module) is also a frequent use case, using the `source` parameter.

For very short files, it is more economic to include the desired content right in the manifest:

```
file { '/etc/modules':
  ensure  => file,
  content => "# Managed by Puppet!\n\nndrbd\n",
}
```



The double quotes allow expansion of escape sequences such as \n.

Another useful capability is managing symbolic links:

```
file { '/etc/apache2/sites-enabled/001-puppet-lore.org':
  ensure => 'link',
  target => '../sites-available/puppet-lore.org',
}
```

The next type that you already know is package, and its typical usage is quite intuitive. Make sure that packages are either installed or removed. A notable use case that you have not yet seen is to use the basic package manager instead of apt or yum/zypper. This is useful if the package is not available from a repository:

```
package { 'haproxy':
  ensure  => present,
  provider => 'dpkg',
  source   => '/opt/packages/haproxy-1.5.1_amd64.deb',
}
```

Your mileage usually increases if you make the effort of setting up a simple repository instead, so that the main package manager can be used after all.

Last but not least, there is a service type, the most important attributes of which you already know. It's worth pointing out that it can serve as a simple shortcut in cases where you don't wish to add a full-fledged init script or something similar. With enough information, the base provider for the service type will manage simple background processes for you:

```
service { 'count-logins':
  provider  => 'base',
  ensure    => 'running',
  binary    => '/usr/local/bin/cnt-logins',
  start     => '/usr/local/bin/cnt-logins --daemonize',
```

```
subscribe => File['/usr/local/bin/cnt-logins'],
}
```

Puppet will not only restart the script if it is not running for some reason, but will also restart it whenever the content of the referenced configuration file changes. This only works if Puppet manages the file content and all changes propagate through Puppet only.



If Puppet changes any other property of the script file (for example, the file mode), that too will lead to a restart of the process.

Let's take a look at some other types you will probably need.

The user and group types

Especially in the absence of central registries such as LDAP, it is useful to be able to manage user accounts on each of your machines. There are providers for all supported platforms, however, the available attributes vary. On Linux, the `useradd` provider is the most common. It allows the management of all fields in `/etc/passwd`, such as `uid` and `shell`, and also group memberships:

```
group { 'proxy-admins':
  ensure => present,
  gid    => 4002,
}
user { 'john':
  ensure      => present,
  uid        => 2014,
  home       => '/home/john'
  managehome => true, # <- adds -m to useradd
  gid        => 1000,
  shell      => '/bin/zsh',
  groups     => [ 'proxy-admins' ],
}
```

As with all resources, Puppet will not only make sure that the user and group exist, but also fix any divergent properties, such as the `home` directory.

Even though the user depends on the group, (because they cannot be added before the group exists) it need not be expressed in the manifest. The user automatically requires all necessary groups, similar to a file auto requiring its parent directory.



Note that Puppet will also happily manage your LDAP user accounts.



It was mentioned earlier that there are different attributes available, depending on the Operating System. Linux (and the `useradd` provider) support setting a password, whereas on HP-UX (using the `hp-ux` provider) the user password cannot be set via Puppet.

In this case, Puppet will only show a warning saying that the user resource type is making use of an unsupported attribute, and will continue managing all other attributes. In other words, using an unsupported attribute in your Puppet DSL code will not break your Puppet run.

The exec resource type

There is one oddball resource type in the Puppet core. Remember our earlier assertion that Puppet is not a specialized scripting engine, but instead, a tool that allows you to model part of your system state in a compelling DSL, and which is capable of altering your system to meet the defined goal. This is why you declare `user` and `group`, instead of invoking `groupadd` and `useradd` in order. You can do this because Puppet comes with support to manage such entities. This is vastly beneficial because Puppet also knows that on different platforms, other commands are used for account management, and that the arguments can be subtly different on some systems.

Of course, Puppet does not have knowledge of all the conceivable particulars of any supported system. Say that you wish to manage an OpenAFS file server. There are no specific resource types to aid you with this. The ideal solution is to exploit Puppet's plugin system and to write your own types and providers so that your manifests can just reflect the AFS-specific configuration. This is not simple though, and also not worthwhile in cases where you only need Puppet to invoke some exotic commands from very few places in your manifest.

For such cases, Puppet ships with the `exec` resource type, which allows the execution of custom commands in lieu of an abstract sync action.

For example, it can be used to unpack a tarball in the absence of a proper package:

```
exec { 'tar cjf /opt/packages/homebrewn-3.2.tar.bz2':
  cwd      => '/opt',
  path     => '/bin:/usr/bin',
  creates  => '/opt/homebrewn-3.2',
}
```

The `creates` parameter is important for Puppet to tell whether the command needs running—once the specified path exists, the resource counts as synchronized. For commands that do not create a telltale file or directory, there are the alternative parameters, `onlyif` and `unless`, to allow Puppet to query the sync state:

```
exec { 'perl -MCPAN -e "install YAML"':
  path    => '/bin:/usr/bin',
  unless  => 'cpan -l | grep -qP ^YAML\\b',
}
```

The query command's exit code determines the state. In the case of `unless`, the `exec` command runs if the query fails. This is how the `exec` type maintains idempotency. Puppet does this automatically for most resource types, but this is not possible for `exec`, because synchronization is defined so arbitrarily. It becomes your responsibility as the user to define the appropriate queries per resource.

Finally, the `exec` type resources are the second notable case of receivers for events using `notify` and `subscribe`:

```
exec { 'apt-get update':
  path      => '/bin:/usr/bin',
  subscribe  => File['/etc/apt/sources.list.d/jenkins.list'],
  refreshonly => true,
}
```

You can even chain multiple `exec` resources in this fashion so that each invocation triggers the next one. However, this is a bad practice and degrades Puppet to a (rather flawed) scripting engine. The `exec` resources should be avoided in favor of regular resources whenever possible. Some resource types that are not part of the core are available as plugins from the Puppet Forge. You will learn more about this topic in *Chapter 5, Extending Your Puppet Infrastructure with Modules*.

Since `exec` resources can be used to perform virtually *any* operation, they are sometimes abused to stand in for more proper resource types. This is a typical antipattern in Puppet manifests. It is safer to regard `exec` resources as the last resort that is only to be used if all other alternatives have been exhausted.



All Puppet installations have the type documentation built into the code, which is printable on command line by using the `puppet describe` command:

```
puppet describe <type> [-s]
```

In case you are unsure whether a type exists, you can tell Puppet `describe` to return a full list of all available resource types:

```
puppet describe --list
```

Let's briefly discuss two more types that are supported out of the box. They allow the management of cron jobs, mounted partitions, and shares respectively, which are all frequent requirements in server operation.

The cron resource type

A cron job mainly consists of a command and the recurring time and date at which to run the command. Puppet models the command and each date particle as a property of a resource with the `cron` type:

```
cron { 'clean-files':
  ensure      => present,
  user        => 'root',
  command     => '/usr/local/bin/clean-files',
  minute      => '1',
  hour        => '3',
  weekday     => [ '2', '6' ],
  environment => 'MAILTO=felix@example.net',
}
```

The `environment` property allows you to specify one or more variable bindings for cron to add to the job.

The mount resource type

Finally, Puppet will manage all aspects of mountable filesystems for you—including their basic attributes such as the source device and mount point, the mount options, and the current state. A line from the `fstab` file translates quite literally to a Puppet manifest:

```
mount { '/media/gluster-data':
  ensure  => 'mounted',
  device  => 'gluster01:/data',
  fstype  => 'glusterfs',
```

```
options => 'defaults,_netdev',
dump   => 0,
pass   => 0,
}
```

For this resource, Puppet will make sure that the filesystem is indeed mounted after the run. Ensuring the unmounted state is also possible, of course; Puppet can also just make sure the entry is present in the `fstab` file, or absent from the system altogether.

Summary

After installing Puppet on your system, you can use it by writing and applying manifests. These manifests are written in Puppet's DSL and contain descriptions of the desired state of your system. Even though they resemble scripts, they should not be considered as such. For one thing, they consist of resources instead of commands. These resources are generally not evaluated in the order in which they have been written. An explicit ordering should be defined through the `require` and `before` metaparameters instead.

Each resource has a number of attributes: parameters and properties. Each property is evaluated in its own right; Puppet detects whether a change to the system is necessary to get any property into the state that is defined in the manifest. It will also perform such changes. This is referred to as **synchronizing** a resource or property.

The ordering parameters, `require` and `before`, are of further importance because they establish dependency of one resource on one or more others. This allows Puppet to skip parts of the catalog if an important resource cannot be synchronized. Circular dependencies must be avoided.

Each resource in the manifest has a resource type that describes the nature of the system entity that is being managed. Some of the types that are used most frequently are `file`, `package`, and `service`. Puppet comes with many types for convenient system management, and many plugins are available to add even more. Some tasks require the use of `exec` resources, but this should be done sparingly.

In the next chapter, we will introduce the master/agent setup.

2

The Master and Its Agents

So far, you have dealt with some concise Puppet manifests that were built to model some very specific goals. By means of the `puppet apply` command, you can use such snippets on any machine in your infrastructure. This is not the most common way of using Puppet though, and this chapter will introduce you to the popular master/agent structure. It's worth noting, however, that applying standalone manifests can always be useful, which are independent of your overall Puppet design.

Under the master/agent paradigm, you will typically install the Puppet agent software on all nodes under your care and make them call the master, which is yet another Puppet installation. The master will compile the appropriate manifests and effectively remotely control the agents. Both the agent and the master authenticate themselves using trusted SSL certificates.

This chapter covers the following topics:

- The Puppet master
- Setting up the Puppet Agent
- Performance considerations
- Using Passenger with Nginx
- SSL troubleshooting

The Puppet Master

Many Puppet-based workflows are centered on the master, which is the central source of configuration data and authority. The master hands instructions to all the computer systems in the infrastructure (where agents are installed). It serves multiple purposes in the distributed system of Puppet components.

The master will perform the following tasks:

- Storing and compiling manifests
- Serving as the SSL certification authority
- Processing reports from the agent machines
- Gathering and storing information about the agents

As such, the security of your master machine is paramount. The requirements for hardening are comparable to those of a Kerberos Key Distribution Center.

During its first initialization, the Puppet master generates the CA certificate. This self-signed certificate will be distributed among and trusted by all the components of your infrastructure. This is why its private key must be protected very carefully. New agent machines request individual certificates, which are signed with the CA certificate.



It's a good idea to include a copy of the CA certificate in your OS-provisioning process so that the agent can establish the authenticity of the master before requesting its individual certificate.

Puppet Master and Puppet Server

The terminology around the master software might be a little confusing. That's because both the terms, **Puppet Master** and **Puppet Server**, are floating around, and they are closely related too. Let's consider some technological background in order to give you a better understanding of what is what.

Puppet's master service mainly comprises a RESTful HTTP API. Agents initiate the HTTPS transactions, with both sides identifying each other using trusted SSL certificates. During the time when Puppet 3 and older versions were current, the HTTPS layer was typically handled by Apache. Puppet's Ruby core was invoked through the Passenger module. This approach offered good stability and scalability.

Puppet Labs has improved upon this standard solution with a specialized software called `puppetserver`. The Ruby-based core of the master remains basically unchanged, although it now runs on JRuby instead of Ruby's own MRI. The HTTPS layer is run by Jetty, sharing the same Java Virtual Machine with the master.

By cutting out some middlemen, `puppetserver` is faster and more scalable than a Passenger solution. It is also significantly easier to set up. Towards the end of this chapter, the two approaches are recapped and visually compared.

Setting up the master machine

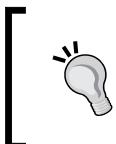
Getting the `puppetserver` software onto a Linux machine is just as simple as the agent package (which you did at the very beginning of *Chapter 1, Writing Your First Manifests*). At the time of writing this, distribution packages were available on Red Hat Enterprise Linux and its derivatives. For Debian and Ubuntu, it was still necessary to rely on Puppet Labs' own repositories to get the software.

A great way to get Puppet Labs packages on any platform is the Puppet Collection. Shortly after the release of Puppet 4, Puppet Labs created this new way of supplying software. This can be considered as a distribution in its own right. Unlike Linux distributions, it does not contain a Kernel, system tools, and libraries. Instead, it comprises various software from the Puppet ecosystem. Software versions that are available from the same Puppet Collection are guaranteed to work well together.

Use the following commands to install `puppetserver` from the first Puppet Collection (PC1) on a Debian 7 machine. (The Collection for Debian 8 has not yet received a `puppetserver` package at the time of writing this.)

```
root@puppetmaster# wget http://apt.puppetlabs.com/puppetlabs-release-pcl-
wheezy.deb
root@puppetmaster# dpkg -i puppetlabs-release-pcl-wheezy.deb
root@puppetmaster# apt-get update
root@puppetmaster# apt-get install puppetserver
```

The `puppetserver` package comprises only the Jetty server and the Clojure API, but the all-in-one `puppet-agent` package is pulled in as a dependency.



The package name, `puppet-agent`, is misleading. This AIO package contains all the parts of Puppet including the master core, a vendored Ruby build, and the several pieces of additional software.

Specifically, you can use the `puppet` command on the master node. You will soon learn how this is useful. However, when using the packages from Puppet Labs, everything gets installed under `/opt/puppetlabs`. It is advisable to make sure that your `PATH` variable always includes the `/opt/puppetlabs/bin` directory so that the `puppet` command is found here.

Regardless of this, once the `puppetserver` package is installed, you can start the master service:

```
root@puppetmaster# service puppetserver start
```

Depending on the power of your machine, the startup can take a few minutes. Once initialization completes, the server will operate very smoothly though. As soon as the master port 8140 is open, your Puppet master is ready to serve requests.



If the service fails to start, there might be an issue with certificate generation. (We observed such issues with some versions of the software.) Check the log file at /var/log/puppetlabs/puppetserver/puppetserver-daemon.log. If it indicates that there are problems while looking up its certificate file, you can work around the problem by temporarily running a standalone master as follows:

```
puppet master --no-daemonize
```

After initialization, you can stop this process. The certificate is available now, and puppetserver should now be able to start as well.

Creating the master manifest

When you used Puppet locally in *Chapter 1, Writing Your First Manifests*, you specified a manifest file that `puppet apply` should compile. The master compiles manifests for many machines, but the agent does not get to choose which source file is to be used—this is completely at the master's discretion. The starting point for any compilation by the master is always the **site manifest**, which can be found in `/opt/puppetlabs/code/environments/production/manifests/`.



The significance of the environments/production part will be investigated in *Chapter 5, Extending Your Puppet Infrastructure with Modules*. In Puppet versions before 4.0, the site manifest is at another location, `/etc/puppet/manifests/site.pp`, and comprises just one file.

Each connecting agent will use all the manifests found here. Of course, you don't want to manage only one identical set of resources on all your machines. To define a piece of manifest exclusively for a specific agent, put it in a node block. This block's contents will only be considered when the calling agent has a matching common name in its SSL certificate. You can dedicate a piece of the manifest to a machine with the name of agent, for example:

```
node 'agent' {
  $packages = [ 'apache2',
    'libapache2-mod-php5',
    'libapache2-mod-passenger', ]
  package { $packages:
```

```
    ensure => 'installed',
    before  => Service['apache2'],
  }
service { 'apache2':
  ensure => 'running',
  enable  => true,
}
}
```

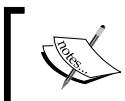
Before you set up and connect your first agent to the master, step back and think about how the master should be addressed. By default, agents will try to resolve the unqualified puppet hostname in order to get the master's address. If you have a default domain that is being searched by your machines, you can use this as a default and add a record for puppet as a subdomain (such as `puppet.example.net`). Otherwise, pick a domain name that seems fitting to you, such as `master.example.net` or `adm01.example.net`. What's important is the following:

- All your agent machines can resolve the name to an address
 - The master process is listening for connections on that address
 - The master uses a certificate with the chosen name as CN or SAN

The mode of resolution depends on your circumstances—the hosts file on each machine is one ubiquitous possibility. The Puppet server listens on all the available addresses by default.

This leaves the task of creating a suitable certificate, which is simple. Configure the master to use the appropriate certificate name and restart the service. If the certificate does not exist yet, Puppet will take the necessary steps to create it. Put the following setting into your `/etc/puppetlabs/puppet/puppet.conf` file on the master machine:

```
[master]  
certname=master.example.net
```



In Puppet versions before 4.0, the default location for the configuration file is `/etc/puppet/puppet.conf`.

Upon its next start, the master will use the appropriate certificate for all SSL connections. The automatic proliferation of SSL data is not dangerous even in an existing setup, except for the certification authority. If the master were to generate a new CA certificate at any point in time, it would break the trust of all existing agents.



Make sure that the CA data is neither lost nor compromised. All previously signed certificates become obsolete whenever Puppet needs to create a new certification authority. The default storage location is `/etc/puppetlabs/puppet/ssl/ca` for Puppet 4.0 and higher, and `/var/lib/puppet/ssl/ca` for older versions.

Inspecting the configuration settings

All the customization of the master's parameters can be made in the `puppet.conf` file. The operating system packages ship with some settings that are deemed sensible by the respective maintainers. Apart from these explicit settings, Puppet relies on defaults that are either built-in or derived from the environment (details on how this works follow in the next chapter):

```
root@puppetmaster # puppet master --configprint manifest  
/etc/puppetlabs/code/environments/production/manifests
```

Most users will want to rely on these defaults for as many settings as possible. This is possible without any drawbacks because Puppet makes all settings fully transparent using the `--configprint` parameter. For example, you can find out where the master manifest files are located.

To get an overview of all available settings and their values, use the following command:

```
root@puppetmaster# puppet master --configprint all | less
```

While this command is especially useful on the master side, the same introspection is available for `puppet apply` and `puppet agent`.

Setting up the Puppet Agent

As was explained earlier, the master mainly serves instructions to agents in the form of catalogs that are compiled from the manifest. You have also prepared a `node` block for your first agent in the master manifest.

Installing the agent software is easy—you did this at the start of *Chapter 1, Writing Your First Manifests*. The plain Puppet package that allows you to apply a local manifest contains all the required parts in order to operate a proper agent.

If you are using Puppet Labs packages, use the instructions from earlier in this chapter. On agent machines, you need not install the `puppetserver` package. Just get `puppet-agent` instead.

After a successful package installation, the following invocation is sufficient for an initial test:

```
root@agent# puppet agent --test
Info: Creating a new SSL key for agent
Error: Could not request certificate: getaddrinfo: Name or service not known
Exiting; failed to retrieve certificate and waitforcert is disabled
```

Puppet first created a new SSL certificate key for itself. For its own name, it picked `agent`, which is the machine's hostname. That's fine for now. An error occurred because the `puppet` name cannot be currently resolved to anything. Add this to `/etc/hosts` so that Puppet can contact the master:

```
root@agent# puppet agent --test
Info: Caching certificate for ca
Info: csr_attributes file loading from /etc/puppetlabs/puppet/csr_attributes.yaml
Info: Creating a new SSL certificate request for agent
Info: Certificate Request fingerprint (SHA256): 52:65:AE:24:5E:2A:C6:17:E2:5D:0A:C9: 86:E3:52:44:A2:EC:55:AE:3D:40:A9:F6:E1:28:31:50:FC:8E:80:69
Error: Could not request certificate: Error 500 on SERVER: Internal Server Error: java.io.FileNotFoundException: /etc/puppetlabs/puppet/ssl/ca/requests/agent.pem (Permission denied)
Exiting; failed to retrieve certificate and waitforcert is disabled
```

Note how Puppet conveniently downloaded and cached the CA certificate. The agent will establish trust based on this certificate from now on.

Puppet created a certificate request and sent it to the master. It then immediately tried to download the signed certificate. This is expected to fail – the master won't just sign a certificate for any request it receives. This behavior is important for proper security.

There is a configuration setting that enables such automatic signing, but users are generally discouraged from using this setting because it allows the creation of arbitrary numbers of signed (and therefore, trusted) certificates to any user who has network access to the master.



To authorize the agent, look for the CSR on the master using the `puppet cert` command:

```
root@puppetmaster# puppet cert --list
"agent" (SHA256) 52:65:AE:24:5E:2A:C6:17:E2:5D:0A:C9:86:E3:52:44:A2:EC:55
:AE: 3D:40:A9:F6:E1:28:31:50:FC:8E:80:69
```

This looks alright, so now you can sign a new certificate for the agent:

```
root@puppetmaster# puppet cert --sign agent
Notice: Signed certificate request for agent
Notice: Removing file Puppet::SSL::CertificateRequest agent at '/etc/
puppetlabs/puppet/ssl/ca/requests/agent.pem'
```



When choosing the action for `puppet cert`, the dashes in front of the option name can be omitted—you can just use `puppet cert list` and `puppet cert sign`.

Now the agent can receive its certificate for its catalog run as follows:

```
root@agent# puppet agent --test
Info: Caching certificate for agent
Info: Caching certificate_revocation_list for ca
Info: Caching certificate for agent
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for agent
Info: Applying configuration version '1437065761'
Notice: Applied catalog in 0.11 seconds
```

The agent is now fully operational. It received a catalog and applied all resources within. Before you read on to learn how the agent usually operates, there is a note that is especially important for the users of Puppet 3.

Remember that you configured the master to use the name `master.example.net` for the master machine earlier in this chapter by setting the `certname` option in the master's `puppet.conf` file. Since this is the common name in the master's certificate, the preceding command will not even work with a Puppet 3.x master. It works with `puppetserver` and Puppet 4 because the default puppet name is now included in the certificate's Subject Alternative Names by default.

It is tidier to not rely on this alias name, though. After all, in production, you will probably want to make sure that the master has a fully qualified name that can be resolved, at least inside your network. You should therefore, add the following to the `main` section of `puppet.conf` on each agent machine:

```
[main]
server=master.example.net
```

In the absence of DNS to resolve this name, your agent will need an appropriate entry in its hosts file or a similar alternative way of address resolution.

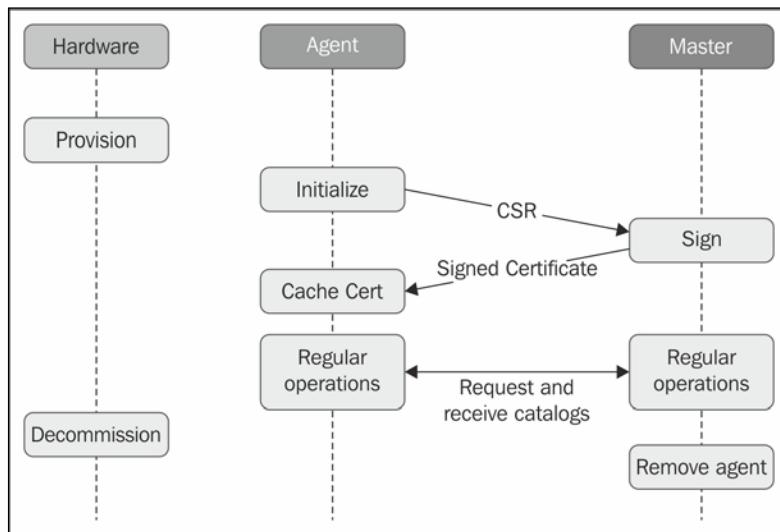
These steps are necessary in a Puppet 3.x setup. If you have been following along with a Puppet 4 agent, you might notice that after this change, it generates a new Certificate Signing Request:

```
root@agent# puppet agent -test
Info: Creating a new SSL key for agent.example.net
Info: csr_attributes file loading from /etc/puppetlabs/puppet/csr_
attributes.yaml
Info: Creating a new SSL certificate request for agent.example.net
Info: Certificate Request fingerprint (SHA256): 85:AC:3E:D7:6E:16:62:BD:2
8:15:B6:18: 12:8E:5D:1C:4E:DE:DF:C3:4E:8F:3E:20:78:1B:79:47:AE:36:98:FD
Exiting; no certificate found and waitforcert is disabled
```

If this happens, you will have to use `puppet cert sign` on the master again. The agent will then retrieve a new certificate.

The agent's life cycle

In a Puppet-centric workflow, you typically want all changes to the configuration of servers (perhaps even workstations) to originate on the Puppet master and propagate to the agents automatically. Each new machine gets integrated into the Puppet infrastructure with the master at its center and gets removed during the decommissioning, as shown in the following diagram:



The very first step—generating a key and a certificate signing request—is always performed implicitly and automatically at the start of an agent run if no local SSL data exists yet. Puppet creates the required data if no appropriate files are found. There will be a short description on how to trigger this behavior manually later in this section.

The next step is usually the signing of the agent's certificate, which is performed on the master. It is a good practice to monitor the pending requests by listing them on the console:

```
root@puppetmaster# puppet cert list
root@puppetmaster# puppet cert sign '<agent fqdn>'
```

From this point on, the agent will periodically check with the master to load updated catalogs. The default interval for this is 30 minutes. The agent will perform a run of a catalog each time and check the sync state of all the contained resources. The run is performed for unchanged catalogs as well, because the sync states can change between runs.



Before you manage to sign the certificate, the agent process will query the master in short intervals for a while. This can avoid a 30 minute delay if the certificate is not ready, right when the agent starts up.

Launching this background process can be done manually through a simple command:

```
root@agent# puppet agent
```

However, it is preferable to do this through the `puppet` system service.

When an agent machine is taken out of active service, its certificate should be invalidated. As is customary with SSL, this is done through **revocation**. The master adds the serial number of the certificate to its certificate revocation list. This list, too, is shared with each agent machine. Revocation is initiated on the master through the `puppet cert` command:

```
root@puppetmaster# puppet cert revoke agent
```



The updated CRL is not honored until the master service is restarted. If security is a concern, this step must not be postponed.

The agent can then no longer use its old certificate:

```
root@agent# puppet agent --test
Warning: Unable to fetch my node definition, but the agent run will
continue:
Warning: SSL_connect SYSCALL returned=5 errno=0 state=unknown state
[...]
Error: Could not retrieve catalog from remote server: SSL_connect SYSCALL
returned=5 errno=0 state=unknown state
[...]
```

Renewing an agent's certificate

Sometimes, it is necessary during an agent machine's life cycle to regenerate its certificate and related data—the reasons can include data loss, human error, or certificate expiration, among others. Performing the regeneration is quite simple: all relevant files are kept at `/etc/puppetlabs/puppet/ssl` (for Puppet 3.x, this is `/var/lib/puppet/ssl`) on the agent machine. Once these files are removed (or rather, the whole `ssl/` directory tree), Puppet will renew everything on the next agent run. Of course, a new certificate must be signed. This requires some preparation—just initiating the request from the agent will fail:

```
root@agent# puppet agent -test
Info: Creating a new SSL key for agent
Info: Caching certificate for ca
Info: Caching certificate for agent.example.net
Error: Could not request certificate: The certificate retrieved from the
master does not match the agent's private key.
Certificate fingerprint: 6A:9F:12:C8:75:C0:B6:10:45:ED:C3:97:24:CC:98:F2:
B6:1A:B5: 4C:E3:98:96:4F:DA:CD:5B:59:E0:7F:F5:E6
```

The master still has the old certificate cached. This is a simple protection against the impersonation of your agents by unauthorized entities. To fix this, remove the certificate from both the master and the agent and then start a puppet run, which will automatically regenerate a certificate.

On the master:

```
puppet cert clean agent.example.net
```

On the agent:

1. On most platforms:

```
find /etc/puppetlabs/puppet/ssl -name agent.example.net.pem
-delete
```

2. On Windows:

```
del "/etc/puppetlabs/puppet/ssl/agent.example.net.pem" /f

puppet agent -t
Exiting; failed to retrieve certificate and waitforcert is disabled
```

Once you perform the cleanup operation on the master, as advised in the preceding output, and remove the indicated file from the agent machine, the agent will be able to successfully place its new CSR:

```
root@puppetmaster# puppet cert clean agent
```

```
Notice: Revoked certificate with serial 18
Notice: Removing file Puppet::SSL::Certificate agent at '/etc/puppetlabs/
puppet/ssl/ca/signed/agent.pem'
Notice: Removing file Puppet::SSL::Certificate agent at '/etc/puppetlabs/
puppet/ssl/certs/agent.pem'
```

The rest of the process is identical to the original certificate creation. The agent uploads its CSR to the master, where the certificate is created through the `puppet cert sign` command.

Running the agent from cron

There is an alternative way to operate the agent. We covered starting one long-running `puppet agent` process that does its work in set intervals and then goes back to sleep. However, it is also possible to have cron launch a discrete agent process in the same interval. This agent will contact the master once, run the received catalog, and then terminate. This has several advantages as follows:

- The agent operating system saves some resources
- The interval is precise and not subject to skew (when running the background agent, deviations result from the time that elapses during the catalog run), and distributed interval skew can lead to thundering herd effects
- Any agent crash or an inadvertent termination is not fatal

Setting Puppet to run the agent from cron is also very easy to do—with Puppet! You can use a manifest such as the following:

```
service { 'puppet': enable => false }
cron { 'puppet-agent-run':
  user      => 'root',
  command   =>
    'puppet agent --no-daemonize --onetime --logdest=syslog',
  minute    => fqdn_rand(60),
  hour      => absent,
}
```

The `fqdn_rand` function computes a distinct minute for each of your agents. Setting the `hour` property to `absent` means that the job should run every hour.

Performance considerations

Operating a Puppet master gives you numerous benefits over just using `puppet apply` on all your machines. This comes at a cost, of course. The master and agents form a server/client relation, and as with most such constructs, the server can become the bottleneck.

The good news is that the Puppet agent is a fat client. The major share of the work—inspecting file contents, interfacing with the package-management subsystem, services subsystem, and much more—is done by the agent. The master *only* has to compile manifests and build catalogs from them. This becomes increasingly complex as you hand over more control to Puppet.

There is one more task your master is responsible for. Many of your manifests will contain file resources that rely on prepared content:

```
file { '/usr/local/etc/my_app.ini':
  ensure => file,
  owner  => 'root',
  group  => 'root',
  source  => 'puppet:///modules/my_app/usr/local/etc/my_app.ini',
}
```

The `source` parameter with a URL value indicates that the file has been pregenerated and placed in a module on the Puppet master (more on modules in *Chapter 5, Extending Your Puppet Infrastructure with Modules*). The agent will compare the local file with the master's copy (by checksum) and download the canonical version, if required. The comparison is a frequent occurrence in most agent runs—you will make Puppet manage a lot of files. The master does not need a lot of resources to make this happen, but it *will* hinder fluent agent operation if the master gets congested.

This can happen for any combination of these reasons:

- The total number of agents is too large
- The agents check in too frequently
- The manifests are too complex
- The Puppet server is not tuned adequately
- The master's hardware resources are insufficient

There are ways to scale your master operation via load balancing, but these are not covered in this book.



Puppet Labs have some documentation on a few advanced approaches at https://docs.puppetlabs.com/guides/scaling_multiple_masters.html.



Tuning puppetserver

puppetserver is a great way to run the master service. It is simple to set up and maintain, and it also has great performance during operation. Starting up can take a little while to initialize everything to that end.

There are only few customizable settings that can impact performance. Seeing as puppetserver runs in the JVM, the most important tuning approach is to scale the heap. A small heap will increase the overhead for garbage collection. Therefore, you should use the `-Xmx` and `-Xms` Java options to allow the JVM to use large parts of your available memory for the mentioned heap.

On Debian, these settings are found in `/etc/default/puppetserver`. It is sensible to pass the same value to both. A dynamic heap has little benefit, because you cannot safely use any saved memory.



For proper puppetserver functionality, it is recommended to have 4 GB of RAM available.

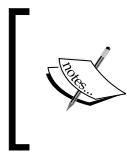


Using Phusion Passenger with Nginx

Some operators don't like running important services in the Java Virtual Machine, for various reasons, such as its memory requirements. As of Puppet 4, alternatives to puppetserver are still available, so the requirement can be avoided.

The best way to run the master without the JVM is a web server with a support for Passenger. In this context, the Puppet master runs as a Rack application. The most common setup comprises the Apache web server and `mod_passenger`. Setting this up is quite straightforward and documentation is plentiful. We will therefore, concentrate on an attractive alternative.

Unfortunately, the Puppet 4 package cannot be made to work with Passenger easily. The best way to achieve this was a manual Puppet installation from the source, at the time of writing this. With Puppet 3.x, Passenger was the default for a long time, and making it work with Nginx is quite simple with the following instructions.



Please note that in Puppet 4, the Rack support is already deprecated. It will likely be removed in a future release. The same holds true for the standalone master that is available through the `puppet master` subcommand (this mode is not covered in this book).

Nginx is a lean and fast web server that is ever increasing in popularity. It can run your Puppet master through Passenger just like Apache, so you don't need to install and run the latter. Unfortunately, the stock version of Nginx cannot run Passenger through a module. The Phusion project supplies packages for the more popular Linux distributions. The following instructions are applicable to Debian:

1. Follow the instructions at https://www.phusionpassenger.com/documentation/Users%20guide%20Nginx.html#install_on_debian_ubuntu in order to install the appropriate Nginx packages.

2. In the `/etc/nginx/nginx.conf` file, uncomment or insert the passenger specific statements:

```
passenger_root /usr/lib/ruby/vendor_ruby/phusion_passenger/
locations.ini;
passenger_ruby /usr/bin/ruby;
```

3. Prepare the Rails root:

```
root@puppetmaster# mkdir -p /etc/puppet/rack/{tmp,public}
root@puppetmaster# install -o puppet -g puppet /usr/share/puppet/
rack/config.ru /etc/puppet/rack
```

4. Create a vhost for Puppet at `/etc/nginx/sites-available/puppetmaster`. Older versions of Passenger use `passenger_set_cgi_param` instead of `passenger_env_var`:

```
server {
    listen 8140;
    server_name master.example.net;
    root /etc/puppet/rack/public;

    ssl on;
    ssl_certificate
        /var/lib/puppet/ssl/certs/master.example.net.pem;
    ssl_certificate_key
        /var/lib/puppet/ssl/private_keys/master.example.net.pem;
    ssl_crl /var/lib/puppet/ssl/ca/ca_crl.pem;
    ssl_client_certificate /var/lib/puppet/ssl/certs/ca.pem;
    ssl_verify_client optional;
    ssl_verify_depth 1;
```

```

passenger_enabled on;
passenger_env_var HTTPS on;
passenger_env_var SSL_CLIENT_S_DN $ssl_client_s_dn;
passenger_env_var SSL_CLIENT_VERIFY $ssl_client_verify;
}

```

5. Enable the vhost and restart Nginx:

```

root@puppetmaster# ln -s ../sites-available/puppetmaster
/etc/nginx/sites-enabled/puppetmaster
root@puppetmaster# /etc/init.d/nginx restart

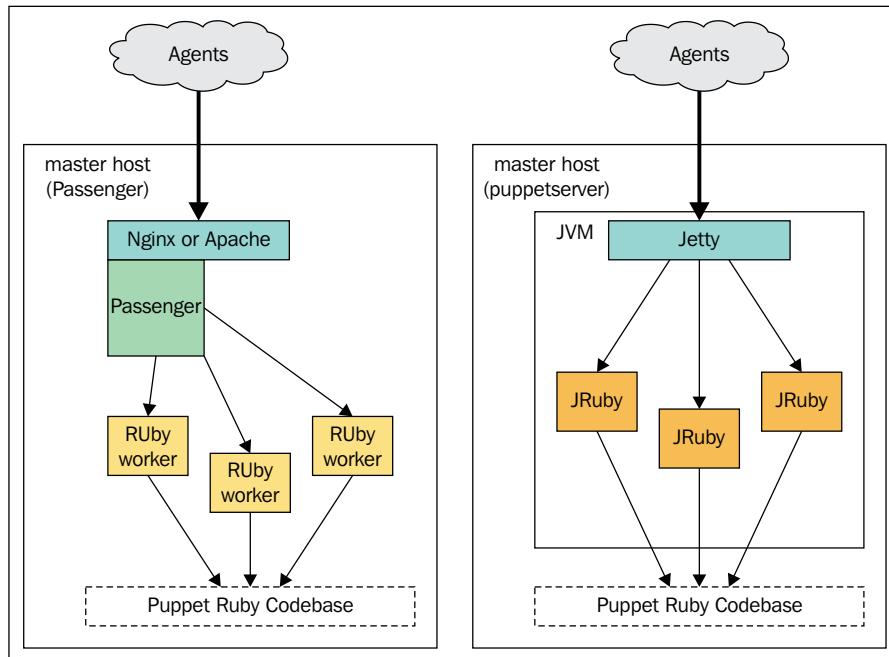
```

Nginx is now running the Puppet master service for you through Passenger. The mentioned configuration is bare boned, and you might wish to extend it for tuning and features.

Comparing Passenger with puppetserver

Both Passenger and puppetserver have their share of complexity. These are much less visible to the user in the case of puppetserver, however. All that's needed is the software package and a Java runtime. The internals are well hidden.

The following diagram highlights the differences:



With puppetserver, both the web service and the Ruby runtimes share a single JVM. This allows a better overall performance, and is easier to set up. All new setups should therefore, prefer puppetserver over Passenger.

Completing the stack with PuppetDB

Whether you choose to use puppetserver or prefer Passenger after all, there is yet another piece of infrastructure that should be considered for each master host. PuppetDB is a specialized database designed to interact with the Puppet master. It mainly comprises a PostgreSQL backend with an API wrapper. The latter was written in Clojure and runs in yet another JVM.

PuppetDB aids the master's secondary task of storing reports and other agent data. It is also necessary for some specific manifest compiler functionality. This is covered in *Chapter 6, Leveraging the Full Toolset of the Language*.

The best way to set up and configure PuppetDB is actually Puppet itself. Since the necessary tools have not yet been introduced, we will postpone this step until *Chapter 6, Leveraging the Full Toolset of the Language*. This is not a problem, because PuppetDB is not essential for basic master operation.



Nevertheless, after finishing this title, you should include PuppetDB into any new master setup, because it allows advanced reporting and introspection.



Troubleshooting SSL issues

Among the most frustrating issues, especially for new users, are problems with the agent's SSL handshake. Such errors are especially troublesome because Puppet cannot always offer very helpful analysis in its logs – the problems occur in the SSL library functions, and the application cannot examine the circumstances.



The online documentation at Puppet Labs has a **Troubleshooting** section that has some advice concerning SSL-related issues as well at <https://docs.puppetlabs.com/guides/troubleshooting.html>.



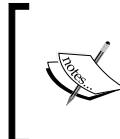
Consider the following output for the `--test` command:

```
root@agent# puppet agent --test
Warning: Unable to fetch my node definition, but the agent run will
continue:
Warning: SSL_connect returned=1 errno=0 state=unknown state: certificate
verify failed: [CRL is not yet valid for /CN=Puppet CA: puppet.example.
net]
```

The agent opines that the CRL it receives from the master is not yet valid. Errors such as these can happen whenever the agent's clock gets reset to a very early date. This can also result from but a slight clock skew, when the CRL has recently been updated through a revocation action on the master. If the system clock on the agent machine returns a time far in the future, it will consider certificates to be expired.

These clock-related issues are best avoided by running an `ntp` service on all Puppet agents and masters.

Errors will generally result if the data in the agent's `$ssldir` becomes inconsistent. This can happen when the agent interacts with an alternate master (a testing instance, for example). The first piece of advice you will most likely receive when asking the community what to do about such problems is to create a new agent certificate from scratch. This works as described in the *The agent's life cycle* section:



Before you start the recovery procedure, make sure that you are logged in to the afflicted agent machine and not the master. Losing the master's SSL data will make it necessary to recreate your complete SSL infrastructure.

1. Remove all the SSL data from the agent machine.
2. Revoke and remove the certificate from the master using `puppet cert clean`.
3. Request and sign a new certificate.

This approach will indeed remedy most issues. Be careful not to leave any old files in the relevant location on the agent machine. If the problems persist, a more involved solution is required. The `openssl` command-line tool is helpful to analyze the certificates and related files. The details of such an analysis are beyond the scope of this book, though.

Summary

You can now set up your own Puppet master, using the sophisticated puppetserver solution. Alternately, you can run your Puppet 3.x master through Apache or Nginx with Passenger. You have successfully signed the certificate for a Puppet agent and can revoke certificates, if required. Using the node blocks in the master manifest, you can describe individual manifests for each distinct agent. Finally, you learned about some things that can go wrong with the SSL-based authentication.

In the next chapter, we will take a look at the inner workings of Puppet in order to give you an understanding of how the Puppet agent adapts itself to its environment. You will also learn how the agent provides feedback to the master, allowing you to create flexible manifests that fit different needs.

3

A Peek under the Hood – Facts, Types, and Providers

So far in this book, you have primarily done practical things – writing manifests, setting up a master, assigning agents, signing certificates, and so forth. Before you are introduced to the missing language concepts that you will need to use Puppet effectively for bigger projects, there is some background that we should cover. Don't worry, it won't be all dry theory – most of the important parts of Puppet are relevant to your daily business.

The topics for this chapter have been hinted at earlier; *Chapter 1, Writing Your First Manifests*, contained a brief description of the type and provider. This and some adjacent topics will be thoroughly explored in the following sections:

- Summarizing systems with Facter
- Understanding the type system
- Substantiating the model with providers
- Putting it all together

Summarizing systems with Facter

Configuration management is quite a dynamic problem. In other words, the systems that need configuration are mostly moving targets. In some situations, system administrators or operators get lucky and work with large quantities of 100 percent uniform hardware and software. In most cases, however, the landscape of servers and other computing nodes is rather heterogeneous, at least in subtle ways. Even in unified networks, there are likely multiple generations of machines or operating systems, with small or larger differences required for their respective configurations.

For example, a common task for Puppet is to handle the configuration of system monitoring. Your business logic will likely dictate warning thresholds for gauges such as the system load value. However, those thresholds can rarely be static. On a two-processor virtual machine, a system load of 10 represents a crippling overload, while the same value can be absolutely acceptable for a busy DBMS server that has cutting edge hardware of the largest dimensions.

Another important factor can be software platforms. Your infrastructure might span multiple distributions of Linux or alternate operating systems such as BSD, Solaris, or Windows, each with different ways of handling certain scenarios. Imagine, for example, that you want Puppet to manage some content from the `fstab` file. On your rare Solaris system, you would have to make sure that Puppet targets the `/etc/vfstab` file instead of `/etc/fstab`.



It is usually not a good idea to interact directly with the `fstab` file in your manifest. This example will be rounded off in the section concerning providers.



Puppet strives to present you with a unified way of managing all your infrastructure. It therefore needs a means to allow your manifests to adapt to different kinds of circumstances on the agent machines. This includes their operating system, hardware layout, and many other details. Keep in mind that, generally, the manifests have to be compiled on the master machine.

There are several conceivable ways to implement a solution for this particular problem. A direct approach would be to use a language construct that allows the master to send a piece of shell script (or other code) to the agent and receive its output in return.



The following is pseudocode, however, there are no back tick expressions in the Puppet DSL:

```
if `grep -c ^processor /proc/cpuinfo` > 2 {  
    $load_warning = 4  
}  
else {  
    $load_warning = 2  
}
```



This solution would be powerful but expensive. The master would need to call back to the agent whenever the compilation process encountered such an expression. Writing manifests that were able to cope with such a command returning an error code would be strenuous, and Puppet would likely end up resembling a quirky scripting engine.



When using `puppet apply` instead of the master, such a feature would pose less of a problem, and it is indeed available in the form of the `generate` function, which works just like the back ticks in the pseudocode mentioned previously. The commands are always run on the compiling node though, so this is less useful with an agent/master than `apply`.

Puppet uses a different approach. It relies on a secondary system called Facter, which has the sole purpose of examining the machine on which it is run. It serves a list of well-known variable names and values, all according to the system on which it runs. For example, an actual Puppet manifest that needs to form a condition upon the number of processors on the agent will use this expression:

```
if $::processorcount > 4 { ... }
```

Facter's variables are called **facts**, and `processorcount` is such a fact. The fact values are gathered by the agent and sent to the master who will use these facts to compile a catalog. All fact names are available in the manifests as variables.



Facts are available to manifests that are used with `puppet apply` too, of course. You can test this very simply:

```
puppet apply -e 'notify { "I am ${::fqdn} and have ${::processorcount} CPUs": }'
```

Accessing and using fact values

You have already seen the use of the `processorcount` fact in an example. In the manifest, each fact value is available as a global variable value. That is why you can just use the `::processorcount` expression where you need it.

 You will often see conventional uses such as `$::processorcount` or `$::ipaddress`. Prefixing the fact name with double colons is highly recommended. The official *Style Guide* at https://docs.puppetlabs.com/guides/style_guide.html#namespacing-variables recommends this. The prefix indicates that you are referring to a variable delivered from Facter. Facter variables are put into the puppet master's top scope.

Some helpful facts have already been mentioned. The `processorcount` fact might play a role for your configuration. When configuring some services, you will want to use the machine's `ipaddress` value in a configuration file or as an argument value:

```
file { '/etc/mysql/conf.d/bind-address':
  ensure  => 'file',
  mode    => '0644',
  content => "[mysqld]\nbind-address=${::ipaddress}\n",
}
```

Besides the hostname, your manifest can also make use of the **fully qualified domain name (FQDN)** of the agent machine.

 The agent will use the value of its `fqdn` fact as the name of its certificate (`clientcert`) by default. The master receives both these values. Note that the agent can override the `fqdn` value to any name, whereas the `clientcert` value is tied to the signed certificate that the agent uses. Sometimes, you will want the master to pass sensitive information to individual nodes. The manifest must identify the agent by its `clientcert` fact and never use `fqdn` or `hostname` instead, for the reason mentioned. An example is shown in the following code:

```
file { '/etc/my-secret':
  ensure  => 'file',
  mode    => '0600',
  owner   => 'root',
  source  => "puppet:///modules/
               secrets/${::clientcert}/key",
}
```

There is a whole group of facts to describe the operating system. Each fact is useful in different situations. The `operatingsystem` fact takes values such as `Debian` or `CentOS`:

```
if $::operatingsystem != 'Ubuntu' {
    package { 'avahi-daemon':
        ensure => absent
    }
}
```

If your manifest will behave identically on RHEL, CentOS, and Fedora (but not on Debian and Ubuntu), you should make use of the `osfamily` fact instead:

```
if $::osfamily == 'RedHat' {
    $kernel_package = 'kernel'
}
```

The `operatingsystemrelease` fact allows you to tailor your manifests to differences between the versions of your OS:

```
if $::operatingsystem == 'Debian' {
    if versioncmp($::operatingsystemrelease, '7.0') >= 0 {
        $ssh_ecdsa_support = true
    }
}
```

Facts such as mac address, the different SSH host keys, fingerprints, and others make it easy to use Puppet for keeping inventory of your hardware. There are a slew of other useful facts. Of course, the collection will not suit every possible need of every user out there. That is why Facter comes readily extendible.

Extending Facter with custom facts

Technically, nothing is stopping you from adding your own fact code right next to the core facts, either by maintaining your own Facter package, or even by deploying the Ruby code files to your agents directly through Puppet management. However, Puppet offers a much more convenient alternative in the form of **custom facts**.

We have still not covered Puppet modules yet. They will be thoroughly introduced in *Chapter 5, Extending Your Puppet Infrastructure with Modules*. For now, just create a Ruby file at `/etc/puppetlabs/code/environments/production/modules/hello_world/lib/facter/hello.rb` on the master machine. Puppet will recognize this as a custom fact of the name, `hello`. (For Puppet 3 or older versions, the path should be `/etc/puppet/modules/hello_world/lib/facter/hello.rb`.)

The inner workings of Facter are very straightforward and goal oriented. There is one block of Ruby code for each fact, and the return value of the block becomes the fact value. Many facts are self-sufficient, but others will rely on the values of one or more basic facts. For example, the method to determine the IP address(es) of the local machine is highly dependent upon the operating system.

The `hello` fact is very simple though:

```
Facter.add(:hello) do
  setcode { "Hello, world!" }
end
```

The return value of the `setcode` block is the string, `Hello, world!`, and you can use this fact as `$::hello` in a Puppet manifest.



Before Facter Version 2.0, each fact had a string value. If a code block returns another value, such as an array or hash, Facter 1.x will convert it to a string. The result is not useful in many cases. For this historic reason, there are facts such as `ipaddress_eth0` and `ipaddress_lo`, instead of (or in addition to) a proper hash structure with interface names and addresses.

It is important for the `pluginsync` option to be enabled on the agent side. This has been the default for a long time and should not require any customization. The agent will synchronize all custom facts whenever checking in to the master. They are permanently available on the agent machine after that. You can then retrieve the `hello` fact from the command line using the following line:

```
# puppet facts | grep hello
```

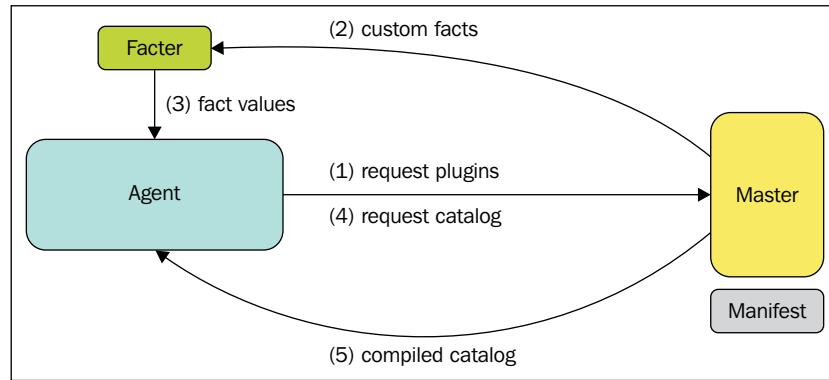
By just invoking the following command without an argument, you request a list of all fact names and values.

```
# puppet facts
```

There is also a `facter` command. It does roughly the same as `puppet facts`, but will only show built in facts, not your custom facts.



In Puppet 3 and earlier, there was no `puppet facts` subcommand. You have to rely on the `facter` CLI (from Facter version 2.x or older) and call `facter -p`, to include custom facts. Facter 3.0 removed this parameter, although it has been announced that newer versions will support it again.



This book will not cover all aspects of Facter's API, but there is one facility that is quite essential. Many of your custom facts will only be useful on Unix-like systems, and others will only be useful on your Windows boxen. You can retrieve such values using a construct like the following:

```

if Facter.value(:kernel) != "windows"
  nil
else
  # actual fact code here
end

```

This would be quite tedious and repetitive though. Instead, you can invoke the `confine` method within the `Facter.add(name) { ... }` block:

```

Facter.add(:msvs_version) do
  confine :kernel => :windows
  setcode do
    # ...
  end
end

```

You can confine a fact to several alternative values as well:

```
confine :kernel => [ :linux, :sunos ]
```

Finally, if a fact does make sense in different circumstances, but requires drastically different code in each respective case, you can add the same fact several times, each with a different set of confine values. Core facts such as `ipaddress` use this often:

```
Facter.add(:ipaddress) do
  confine :kernel => :linux
  ...
end
Facter.add(:ipaddress) do
  confine :kernel => %w{FreeBSD OpenBSD Darwin DragonFly}
  ...
end
...

```

You can confine facts based on any combination of other facts, not just `kernel`. It is a very popular choice though. The `operatingsystem` or `osfamily` facts can be more appropriate in certain situations. Technically, you can even confine some of your facts to certain `processorcount` values and so forth.

Simplifying things using external facts

If writing and maintaining Ruby code is not desirable in your team for any reason, you might prefer to use an alternative that allows shell scripts, or really, any kind of programming language, or even static data with no programming involved at all. Facter allows this in the form of **external facts**.

Creating an external fact is similar to the process for regular custom facts, with the following distinctions:

- Facts are produced by standalone executables or files with static data, which the agent must find in `/etc/puppetlabs/facter/facts.d/`
- The data is not just a string value, but an arbitrary number of `key=value` pairs instead

The data need not use the `ini` file notation style – the key/value pairs can also be in the YAML or JSON format. The following external facts hold the same data:

```
# site-facts.txt
workgroup=CT4Site2
domain_psk=nm56DxLp%
```

The facts can be written in the YAML format in the following way:

```
# site-facts.yaml
workgroup: CT4Site2
domain_psk: nm56DxLp%
```

In the JSON format, facts can be written as follows:

```
# site-facts.json
{ 'workgroup': 'CT4Site2', 'domain_psk': 'nm56DxLp%' }
```

The deployment of the external facts works simply through `file` resources in your Puppet manifest:

```
file { '/etc/puppetlabs/facter/facts.d/site-facts.yaml':
  ensure => 'file',
  source => 'puppet:///...',
```

 With newer versions of Puppet and Facter, external facts will be automatically synchronized, just like custom facts if they are found in `facts.d/*` in any module (for example, `/etc/puppetlabs/code/environments/production/modules/hello_world/facts.d/hello.sh`). This is not only more convenient, but has a large benefit; when Puppet must fetch an external fact through a `file` resource instead, its fact value(s) are not available while the catalog is being compiled. The `pluginsync` mechanism, on the other hand, makes sure that all synced facts are available before manifest compilation starts.

When facts are not static and cannot be placed in a `.txt` or `YAML` file, you can make the file executable and add a shebang instead. It will usually be a shell script, but the implementation is of no consequence; it is just important that properly formatted data is written to the standard output. You can simplify the `hello` fact this way, in `/etc/puppetlabs/code/environments/production/modules/hello_world/facts.d/hello.sh`:

```
#!/bin/sh

echo hello=Hello, world\!
```

For executable facts, the `ini` styled key=value format is the only supported one. `YAML` or `JSON` are not eligible in this context.

 Facter 2 introduced structured facts. Structured facts return an array or a hash. In older Puppet versions (prior to 3.4), structured facts have to be enabled in `puppet.conf` by setting `stringify_facts` to false. This setting is default for Puppet 4.0 and later versions.

Goals of Facter

The whole structure and philosophy of Facter serves the goal of allowing for platform-agnostic usage and development. The same collection of facts (roughly) is available on all supported platforms. This allows Puppet users to keep a coherent development style through manifests for all those different systems.

Facter forms a layer of abstraction over the characteristics of both hardware and software. It is an important piece of Puppet's platform-independent architecture. Another piece that was mentioned before is the type and provider subsystem. Types and providers are explored in greater detail in the following sections.

Understanding the type system

Being one of the cornerstones of the Puppet model, resources were introduced quite early in *Chapter 1, Writing Your First Manifests*. Remember how each resource represents a piece of state on the agent system. It has a resource type, a name (or a title), and a list of attributes. An attribute can either be property or parameter. Between the two of them, properties represent distinct pieces of state, and parameters merely influence Puppet's actions upon the property values.

Let's examine resource types in more detail and understand their inner workings. This is not only important when extending Puppet with resource types of your own (which will be demonstrated in *Chapter 5, Extending Your Puppet Infrastructure with Modules*). It also helps you anticipate the action that Puppet will take, given your manifest, and get a better understanding of both the master and the agent.

First, we take a closer look at the operational structure of Puppet, with its pieces and phases. The agent performs all its work in discreet **transactions**. A transaction is started under any of the following circumstances:

- The background agent process activates and checks in to the master
- An agent process is started with the `--onetime` or `--test` options
- A local manifest is compiled using `puppet apply`

The transaction always passes several stages. They are as follows:

1. Gathering fact values to form the actual catalog request.
2. Receiving the compiled catalog from the master.
3. Prefetching of current resource states.
4. Validation of the catalog's content.
5. Synchronization of the system with the property values from the catalog.

Facter was explained in the previous section. The resource types become important during compilation and then throughout the rest of the agent transaction. The master loads all resource types to perform some basic checking—it basically makes sure that the types of resources it finds in the manifests do exist and that the attribute names fit the respective type.

The resource type's life cycle on the agent side

Once the compilation has succeeded, the master hands out the catalog and the agent enters the catalog validation phase. Each resource type can define some Ruby methods that ensure that the passed values make sense. This happens on two levels of granularity: each attribute can validate its input value, and then the resource as a whole can be checked for consistency.

One example for attribute value validation can be found in the `ssh_authorized_key` resource type. A resource of this type fails if its `key` value contains a whitespace character, because SSH keys cannot comprise multiple strings.

Validation of whole resources happens with the `cron` type, for example. It makes sure that the `time` fields make sense together. The following resource would not pass, because special times, such as `midnight`, cannot be combined with numeric fields:

```
cron { 'invalid-resource':
  command => 'apt-get update',
  special => 'midnight',
  weekday => [ '2', '5' ],
}
```

Another task during this phase is the transformation of input values to more suitable internal representations. The resource type code refers to this as a `munge` action. Typical examples of munging are the removal of leading and trailing whitespace from string values, or the conversion of array values to an appropriate string format—this can be a comma-separated list, but for search paths, the separator should be a colon instead. Other kinds of values will use different representations.

Next up is the prefetching phase. Some resource types allow the agent to create an internal list of resource instances that are present on the system. These types are referred to as being enumerable types. For example, this is possible (and makes sense) for installed packages—Puppet can just invoke the package manager to produce the list. For other types, such as `file`, this would not be prudent. Creating a list of all reachable paths in the whole filesystem can be arbitrarily expensive, depending on the system on which the agent is running.

The prefetching can be simulated by running `puppet resource <resource type> <title>` on the command line as follows:

```
# puppet resource user root
user { 'root':
  ensure      => 'present',
  comment     => 'root',
  gid         => '0',
  home        => '/root',
  password    => '$6$17/7FtU/$TvYED
tFgGr0SaS7x0VloWXVTqQxxDUGH.eBKJ7bgHJ.
hdoc03Xrvvm2ru0HFKpu1QSpVW/7o.rLdk/9MZANEgt',
  password_max_age => '99999',
  password_min_age => '0',
  shell        => '/bin/bash',
  uid          => '0',
}
```



Finally, the agent starts walking its internal graph of interdependent resources. Each resource is brought in sync, if necessary. This happens separately for each individual property, for the most part.

The `ensure` property, for types that support it, is a notable exception. It is expected to manage all other properties on its own—when a resource is changed from absent to present through its `ensure` property (in other words, the resource is getting newly created), this action should bring all other properties in sync as well.



There are some notable aspects of the whole agent process. For one, attributes are handled independently. Each can define its own methods for the different phases. There are quite a number of hooks, which allow a resource type author to add a lot of flexibility to the model.

For aspiring type authors, skimming through the core types can be quite inspirational. You will be familiar with many attributes; using them in your manifests and studying their hooks will offer quite some insight.



It is also worth noting that the whole validation process is performed by the agent, not the master. This is beneficial in terms of performance. The master saves a lot of work, which gets distributed to the network of agents (which scales with your needs automatically).

Substantiating the model with providers

At the start of this chapter, you learned about Facter and how it works like a layer of abstraction over the supported platforms. This unified information base is one of Puppet's most important means to achieve its goal of operating system independence. Another one is the DSL, of course. Finally, Puppet also needs a method to transparently adapt its behavior to the respective platform on which each agent runs.

In other words, depending on the characteristics of the computing environment, the agent needs to switch between different implementations for its resources. This is not unlike object-oriented programming – the type system provides a unified interface, not unlike an abstract base class. The programmer need not worry what specific class is being referenced, as long as it correctly implements all the required methods. In this analogy, Puppet's providers are the concrete classes that implement the abstract interface.

For a practical example, look at package management. Different flavors of UNIX-like operating systems have their own implementation. The most prevalent Puppet platforms use `apt` and `yum`, respectively, but can (and sometimes must) also manage their packages through `dpkg` and `rpm`. Other platforms use tools such as `emerge`, `zypper`, `fink`, and a slew of other things. There are even packages that exist apart from the operating system software base, handled through `gem`, `pip`, and other language-specific package management tools. For each of these management tools, there is a provider for the package type.

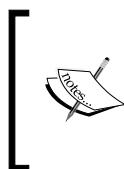
Many of these tools allow the same set of operations – installing and uninstalling a package, and updating a package to a specific version. The latter is not universally possible though. For example, `dpkg` can only ever install the local package that is specified on the command line, with no other version to choose.

There are also some distinct features that are unique to specific tools, or are supported by only a few. Some management systems can hold packages at specific versions. Some use different states for uninstalled versus purged packages. Some have a notion of virtual packages. The list of examples can go on and on.

Because of this potential diversity (which is not limited to package management systems), Puppet providers can opt for **features**. The set of features is resource-type specific. All providers for a type can support one or more of the same group of features. For the package type, there are features such as `versionable`, `purgeable`, `holdable`, and so forth. You can set `ensure => purged` on any package resource like so:

```
package { 'haproxy':  
    ensure => 'purged'  
}
```

However, if you are managing the `haproxy` package through `rpm`, Puppet will fail to make any sense of it because `rpm` has no notion of a purged state, and therefore, the `purgeable` feature is missing from the `rpm` provider. Trying to use an unsupported feature will usually produce an error message. Some attributes, such as `install_options`, might just result in a warning by Puppet instead.

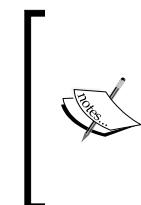


The official documentation on the Puppet Labs website holds a complete list of the core resource types and all their built-in providers, along with the respective feature matrices. It is very easy to find suitable providers and their capabilities. The documentation can be found at <https://docs.puppetlabs.com/references/latest/type.html>.

Resource types with generic providers

There are some resource types that use no providers, but they are rare among the core types. Most of the interesting management tasks that Puppet makes easy just work differently among operating systems, and providers enable this in a most elegant fashion.

Even for straightforward tasks that are the same on all platforms, there might be a provider. For example, there is a `host` type to manage entries in the `/etc/hosts` file. Its syntax is universal, so the code can technically just be implemented in the type. However, there are actual abstract base classes for certain kinds of providers in the Puppet code base. One of them makes it very easy to build providers that edit files, if those files consist of single-line records with ordered fields. Therefore, it makes sense to implement a provider for the `host` type and base it on this provider class.



For the curious, this is what a host resource looks like:

```
host { 'puppet':
  ip           => '10.144.12.100',
  host_aliases => [ 'puppet.example.net', 'master' ],
}
```

Summarizing types and providers

Puppet's resource types and their providers work together to form a solid abstraction layer over software configuration details. The type system is an extendable basis for Puppet's powerful DSL. It forms an elaborate interface for the polymorphous provider layer.

The providers flexibly implement the actual management actions that Puppet is supposed to perform. They map the necessary synchronization steps to commands and system interactions. Many providers cannot satisfy every nuance that the resource type models. The feature system takes care of these disparities in a transparent fashion.

Putting it all together

Reading this far, you might have gotten the impression that this chapter is a rather odd mix of topics. While types and providers do belong closely together, the whole introduction to Facter might seem out of place in this context. This is deceptive, however; facts do play a vital role in the type/provider structure. They are essential for Puppet to make good choices among providers.

Let's look at an example from the *Extending Facter with custom facts* section once more. It was about `fstab` entries and the difference of Solaris, which uses `/etc/vfstab` instead of `/etc/fstab`. That section suggested a manifest that adapts according to a fact value. As you have learned, Puppet has a resource type to manage `fstab` content: the `mount` type. However, for the small deviation of a different file path, there is no dedicated `mount` provider for Solaris. There is actually just one provider for all platforms, but on Solaris, it behaves differently. It does this by resolving Facter's `osfamily` value. The following code example was adapted from the actual provider code:

```
case Facter.value(:osfamily)
when "Solaris"
  fstab = "/etc/vfstab"
else
  fstab = "/etc/fstab"
end
```

In other cases, Puppet should use thoroughly different providers on different platforms, though. Package management is a classic example. On a Red Hat-like platform, you will want Puppet to use the `yum` provider in virtually all cases. It can be sensible to use `rpm`, and even `apt` might be available. However, if you tell Puppet to make sure a package is installed, you expect it to install it using `yum`, if necessary.

This is obviously a common theme. Certain management tasks need to be performed in different environments, with very different toolchains. In such cases, it is quite clear which provider would be best suited. To make this happen, a provider can declare itself the default if a condition is met. In the case of `yum`, it is the following:

```
defaultfor :operatingsystem => [:fedora, :centos, :redhat]
```

The conditions are based around fact values. If the `operatingsystem` value for a given agent is among the listed, `yum` will consider itself the default package provider.



The `operatingsystem` and `osfamily` facts are the most popular choices for such queries in providers, but any fact is eligible.

In addition to marking themselves as being default, there is more filtering of providers that relies on fact values. Providers can also confine themselves to certain combinations of values. For example, the `yum` alternative, `zypper`, confines itself to SUSE Linux distributions:

```
confine :operatingsystem => [:suse, :sles, :sled, :opensuse]
```

This provider method works just like the `confine` method in Facter, which was discussed earlier in this chapter. The provider will not even be seen as valid if the respective facts on the agent machine have none of the white-listed values.



If you find yourself looking through code for some core providers, you will notice confinement (and even declaring default providers) on feature values, although there is no Facter fact of that name. These features are not related to provider features either. They are from another layer of introspection similar to Facter, but hardcoded into the Puppet agent. These agent features are a number of flags that identify some system properties that need not be made available to manifests in the form of facts. For example, the `posix` provider for the `exec` type becomes the default in the presence of the corresponding feature:

```
defaultfor :feature => :posix
```

You will find that some providers forgo the `confine` method altogether, as it is not mandatory for correct agent operation. Puppet will also identify unsuitable providers when looking for their necessary operating system commands. For example, the `pw` provider for certain BSD flavors does not bother with a `confine` statement. It only declares its one required command:

```
commands :pw => "pw"
```

Agents that find no `pw` binary in their search path will not try and use this provider at all.

This concludes the little tour of the inner workings of types and providers with the help of Facter. For a complete example of building a provider for a type, and using the internal tools that you have now learned about, you can refer to *Chapter 5, Extending Your Puppet Infrastructure with Modules*.

Summary

Puppet gathers information about all agent systems using Facter. The information base consists of a large number of independent bits, called facts. Manifests can query the values of those facts to adapt to the respective agents that trigger their compilation. Puppet also uses facts to choose among providers the work horses that make the abstract resource types functional across the wide range of supported platforms.

The resource types not only completely define the interface that Puppet exposes in the DSL, they also take care of all the validation of input values, transformations that must be performed before handing values to the providers, and other related tasks.

The providers encapsulate all knowledge of actual operating systems and their respective toolchains. They implement the functionality that the resource types describe. The Puppet model's configurations apply to platforms, which vary from one another, so not every facet of every resource type can make sense for all agents. By exposing only the supported features, a provider can express such limitations.

After this in-depth look at the internal details, let's tackle more practical concerns again. The following chapters will cover the tools needed to build complex and advanced manifests of all scales.

4

Modularizing Manifests with Classes and Defined Types

At this point, you have already performed some production-grade tasks with Puppet. You learned how to write standalone manifests and then invoke `puppet apply` to put them to use. While setting up your first Puppet master and agent, you created a simple example for a node manifest on the master. In a node '`<hostname>`' block, you created the equivalent of a manifest file. This way, the Puppet master used just this manifest for the specified agent node.

While this is all useful, and of essential importance, it will obviously not suffice for daily business. By working with node blocks that contain sets of resources, you will end up performing lots of copy and paste operations for similar nodes, and the whole construct will become unwieldy very quickly.

This is an unnatural approach to developing Puppet manifests. Despite the great differences to many other languages that you might be familiar with, the Puppet DSL is a programming language. Building manifests merely from node blocks and resources would be like writing C with no functions except `main`, or Ruby without any classes of your own.

The manifests that you can write with the means that are already at your disposal are not flat—you learned about common control structures such as `if` and `case`. Your manifests can use these to adapt to various circumstances on the agent, by querying the values of Facter facts and branching in accordance with the results.

However, these constructs should be complemented by the language tools to create reusable units of manifest code, similar to functions or methods in procedural languages. This chapter introduces these concepts through the following topics:

- Introducing classes and defined types
- Structured design patterns
- Including classes from defined types
- Nesting definitions in classes
- Establishing relationships among containers
- Making classes more flexible through parameters

Introducing classes and defined types

Puppet's equivalents to methods or functions are twofold: there are **classes** on one hand and **defined types** (also just referred to as **defines**) on the other.



You will find that the function analogy is a bit weak for classes, but fits defined types quite well.



They are similar at first glance, in that they both hold a chunk of reusable manifest code. There are big differences in the way each is used though. Let's take a look at classes first.

Defining and declaring classes

A Puppet class can be considered to be a container for resources. It is defined once and selected by all nodes that need to make use of the prepared functionality. Each class represents a well-known subset of a system's configuration.

For example, a classic use case is a class that installs the Apache web server and applies some basic settings. This class will look like the following:

```
class apache {
  package { 'apache2':
    ensure => 'installed',
  }
  file { '/etc/apache2/apache2.conf':
    ensure => 'file',
    source =>
```

```

    'puppet:///modules/apache/etc/apache2/apache2.conf',
    require => Package['apache2'],
}
service { 'apache2':
    enable    => true,
    subscribe => File['/etc/apache2/apache2.conf',
}
}

```

All web server nodes will make use of this class. To this end, their manifests need to contain a simple statement:

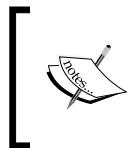
```
include apache
```

This is referred to as **including** a class, or **declaring** it. If your apache class is powerful enough to do all that is needed, this line might fully comprise a node block's content:

```

node 'webserver01' {
    include apache
}

```



In your own setup, you will likely not write your own Apache class. You can use open source classes that are available through **Puppet modules**. *Chapter 5, Extending Your Puppet Infrastructure with Modules*, will give you all the details.

This already concludes our tour of classes in a nutshell. There is yet some more to discuss, of course, but let's take a look at defined types before that.

Creating and using defined types

A defined type can be imagined like a blueprint for a piece of manifest. Like a class, it mainly consists of a body of the manifest code. However, a defined type takes arguments and makes their values available in its body as local variables.

Here is another typical example of a defined type, the Apache virtual host configuration:

```

define virtual_host(
    String $content,
    String[3,3] $priority = '050'
) {

```

```
file { "/etc/apache2/sites-available/${name}":
  ensure  => 'file',
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => $content
}
file { "/etc/apache2/sites-enabled/${priority}-${name}":
  ensure => 'link',
  target => "../sites-available/${name}";
}
}
```



Data types such as `String` are available since Puppet 4. In Puppet 3 and earlier, you would have just skipped these; all variables used to be untyped.

This code might still seem pretty cryptic. It will get clearer in the context of how it is actually used from other places in your manifest; the following code shows you how:

```
virtual_host { 'example.net':
  content => file('apache/vhosts/example.net')
}
virtual_host{ 'fallback':
  priority => '999',
  content  => file('apache/vhosts/fallback')
}
```

This is why the construct is called a defined type—you can now place what appear to be resources in your manifest, but you really call your own manifest code construct.

When declaring multiple resources of the same type, like in the preceding code, you can do so in a single block and separate them with a semicolon:

```
virtual_host {
  'example.net':
    content => 'foo';
  'fallback':
    priority => '999',
    content => ...,
}
```

The official Style Guide forbids this syntax, but it can make manifests more readable and maintainable in some cases.

The `virtual_host` type takes two arguments: the `content` argument is mandatory and is used verbatim in the configuration file resource. Puppet will synchronize that file's content to what is specified in the manifest. The `priority` argument is optional and its value becomes the file name prefix. If omitted, the respective virtual host definition uses the default priority of 050.

Both parameters of this example type are of the `String` type. For details about Puppet's variable type system, see *Chapter 7, New Features from Puppet 4*. It suffices to say that you can restrict parameters to certain value types. This is optional, however. You can omit the type name, and Puppet will accept any value for the parameter in question.

Also, each defined type can implicitly refer to the name (or title) by which it was called. In other words, each instance of your define gets a name, and you can access it through the `$name` or `$title` variable.

There are a few other *magic* variables that are available in the body of a defined type. If a resource of the defined type is declared with a metaparameter such as `require => ...`, its value can be accessed through the `$require` variable in the body. The variable value remains empty if the metaparameter is not used. This works for all metaparameters, such as `before`, `notify`, and all the others, but you will likely never need to make use of this. The metaparameters automatically do the right thing.

Understanding and leveraging the differences

The respective purposes of Puppet's class and defined type are very specific and they usually don't overlap.

The class declares resources and properties that are in some way centric to the system. A class is a finalized description of one or sometimes more aspects of your system as a whole. Whatever the class represents, it can only ever exist in one form; to Puppet, each class is implicitly a singleton, a fixed set of information that either applies to your system (the class is included), or not.

The typical resources you will encapsulate in a class for convenient inclusion in a manifest are as follows:

- One or more packages that should be installed (or removed)
- A specific configuration file in /etc
- A common directory, needed to store scripts or configs for many subsystems
- Cron jobs that should be mostly identical on all applicable systems

The define is used for all things that exist in multiple instances. All aspects that appear in varying quantities in your system can possibly be modeled using this language construct. In this regard, the define is very similar to the full-fledged resource it mimics with its declaration syntax. Some of the typical contents of defined types are:

- Files in a conf.d style directory
- Entries in an easily parseable file such as /etc/hosts
- Apache virtual hosts
- Schemas in a database
- Rules in a firewall

The class's singleton nature is especially valuable because it prevents clashes in the form of multiple resource declarations. Remember that each resource must be unique to a catalog. For example, consider a second declaration of the Apache package:

```
package { 'apache2': }
```

This declaration can be anywhere in the manifest of one of your web servers (say, right in the node block, next to `include apache`); this additional declaration will prevent the successful compilation of a catalog.



The reason for the prevention of a successful compilation is that Puppet currently cannot make sure that both declarations represent the same target state, or can be merged to form a composite state. It is likely that multiple declarations of the same resource get in a conflict about the desired value of some property (for example, one declaration might want to ensure that a package is absent, while the other needs it to be present).

The virtue of the class is that there can be an arbitrary number of `include` statements for the same class strewn throughout the manifest. Puppet will commit the class's contents to the catalog exactly once.

Note that the uniqueness constraint for resources applies to defined types. No two instances of your own define can share the same name. Using a name twice or more produces a compiler error:



```
virtual_host { 'wordpress':
  content  => file(...),
  priority => '011',
}
virtual_host { 'wordpress':
  content  => '# Dummy vhost',
  priority => '600',
}
```

Structured design patterns

Your knowledge of classes and defined types is still rather academic. You have learned about their defining aspects and the syntax to use them, but we have yet to give you a feeling of how these concepts come to bear in different real-life scenarios.

The following sections will present an overview of what you can do with these language tools.

Writing comprehensive classes

Many classes are written to make Puppet perform momentous tasks on the agent platform. Of these, the Apache class is probably one of the more modest examples. You can conceive a class that can be included from any machine's manifest and make sure that the following conditions are met:

- The firewalling software is installed and configured with a default ruleset
- The malware detection software is installed
- Cron jobs run the scanners in set intervals
- The mailing subsystem is configured to make sure the cron jobs can deliver their output

There are two general ways you can go about the task of creating a class of this magnitude. It can either become what one might call a **monolithic** implementation, a class with a large body that comprises all resources that work together to form the desired security baseline. On the other hand, you could aim for a **composite** design, with few resources (or none at all) in the class body, and a number of `include` statements for simpler classes instead. The functionality is compartmentalized, and the central class acts as a collector.

We have not yet touched on the ability of classes to include other classes. That's because it's quite simple. The body of a class can comprise almost any manifest, and the `include` statement is no exception. Among the few things that cannot appear in a class are `node` blocks.

Adding some life to the descriptions, this is how the respective classes will roughly look like:

```
class monolithic_security {  
    package { [ 'iptables', 'rkhunter', 'postfix' ]:  
        ensure => 'installed';  
    }  
    cron { 'run-rkhunter':  
        ...  
    }  
    file { '/etc/init.d/iptables-firewall':  
        source => ...  
        mode   => 755  
    }  
}
```

```
file { '/etc/postfix/main.cf':
  ensure  => 'file',
  content => ...
}
service { [ 'postfix', 'iptables-firewall' ]:
  ensure => 'running',
  enable  => true
}
}

class divided_security {
  include iptables_firewall
  include rkhunter
  include postfix
}
```

When developing your own functional classes, you should not try to pick either of these extremes. Most classes will end up anywhere on the spectrum in between. The choice can be largely based on your personal preference. The technical implications are subtle, but these are the respective drawbacks:

- Consequently aiming for monolithic classes opens you up to resource clashes, because you take almost no advantage of the singleton nature of classes
- Splitting up classes too much can make it difficult to impose order and distribute refresh events—you can refer to the *Establishing relationships among containers* section later in this chapter

Neither of these aspects is of critical importance at most times. The case-by-case design choices will be based on each author's experience and preference. When in doubt, lean towards composite designs at first.

Writing component classes

There is another common use case for classes. Instead of filling a class with lots of aspects that work together to achieve a complex goal, you can also limit the class to a very specific purpose. Some classes will contain but one resource. The class wraps the resource, so to speak.

This is useful for resources that are needed in different contexts. By wrapping them away in a class, you can make sure that those contexts do not create multiple declarations of the same resource.

For example, the `netcat` package can be useful to firewall servers, but also to web application servers. There is probably a `firewall` class and an `appserver` class. Both declare the `netcat` package:

```
package { 'netcat':
  ensure => 'installed'
}
```

If any server ever has both roles (this might happen for budget reasons or in other unforeseen circumstances), it is a problem; when both the `firewall` and `appserver` classes are included, the resulting manifest declares the `netcat` package twice. This is forbidden. To resolve this situation, the package resource can be wrapped in a `netcat` class, which is included by both the `firewall` and `appserver` classes:

```
class netcat {
  package { 'netcat':
    ensure => 'installed'
  }
}
```

Let's consider another typical example for component classes that ensures the presence of some common file path. Assume your IT policy requires all custom scripts and applications to be installed in `/opt/company/bin`. Many classes, such as `firewall` and `appserver` from the previous example, will need some relevant content there. Each class needs to make sure that the directories exist before a script can be deployed inside it. This will be implemented by including a component class that wraps the `file` resources of the directory tree:

```
class scripts_directory {
  file { [ '/opt/company/', '/opt/company/bin' ]:
    ensure => 'directory',
    owner  => 'root',
    group  => 'root',
    mode   => '0644',
  }
}
```

The component class is a pretty precise concept. However, as you have seen in the previous section about the more powerful classes, the whole range of possible class designs forms a fine-grained scale between the presented examples. All manifests you write will likely comprise more than a few classes. The best way to get a feeling for the best practices is to just go ahead and use classes to build the manifests you need.

 The terms **comprehensive** class and **component** class are not official Puppet language, and the community does not use them to communicate design practices. We chose them arbitrarily to describe the ideas we laid out in these sections. The same holds true for the descriptions of the use cases for defined types, which will be seen in the next sections.

Next, let's look at some uses for defined types.

Using defined types as resource wrappers

For all their apparent similarity to classes, defined types are used in different ways. For example, the component class was described as *wrapping a resource*. This is accurate in a very specific context—the wrapped resource is a singleton, and it can only appear in one form throughout the manifest.

When wrapping a resource in a defined type instead, you end up with a variation on the respective resource type. The manifest can contain an arbitrary number of instances of the defined type, and each will wrap a distinct resource.

 For this to work, the name of the resource that is declared in the body of the defined type must be dynamically created. It is almost always the \$name variable of the respective defined type instance, or a value derived from it.

Here is yet another typical example from the many manifests out there: most users who make use of Puppet's file serving capabilities will want to wrap the `file` type at some point so that the respective URL need not be typed for each file:

```
define module_file(String $module) {
  file { $name:
    source => "puppet:///modules/${module}/${name}"
  }
}
```

This makes it easy to get Puppet to sync files from the master to the agent. The master copy must be properly placed in the named modules on the master:

```
module_file { '/etc/ntp.conf':
  module => 'ntp'
}
```

This resource will make Puppet retrieve the `ntp.conf` file from the `ntp` module. The preceding declaration is more concise and less redundant than the fully written file resource with the Puppet URL (especially for the large number of files you might need to synchronize), which would resemble the following:

```
file { '/etc/ntp.conf':
  source => 'puppet:///modules/ntp/etc/ntp.conf':
}
```

For a wrapper such as `module_file`, which will likely be used very widely, you will want to make sure that it supports all attributes of the wrapped resource type. In this case, the `module_file` wrapper should accept all `file` attributes. For example, this is how you add the `mode` attribute to the wrapper type:

```
define module_file(
  String $module,
  Optional[String] $mode = undef
) {
  if $mode != undef {
    File { mode => $mode }
  }
  file { $name:
    source => "puppet:///modules/${module}/${name}"
  }
}
```

The `File { ... }` block declares some default values for all `file` resource attributes in the same scope. The `undef` value is similar to Ruby's `nil`, and is a convenient parameter default value, because it is very unlikely that a user will need to pass it as an actual value for the wrapped resource.

You can employ the override syntax instead of the default syntax as well:

```
File[$name] { mode => $mode }
```

This makes the intent of the code slightly more obvious, but is not necessary in the presence of just one `file` resource. *Chapter 6, Leveraging the Full Toolset of the Language*, holds more information about overrides and defaults.

Using defined types as resource multiplexers

Wrapping single resources with a defined type is useful, but sometimes you will want to add functionality beyond the resource type you are wrapping. At other times, you might wish for your defined type to unify a lot of functionality, just like the comprehensive classes from the beginning of the section.

For both scenarios, what you want to have is multiple resources in the body of your defined type. There is a classic example for this as well:

```
define user_with_key(
  String $key,
  Optional[String] $uid = undef,
  String $group = 'users'
) {
  user { $title:
    ensure      => present
    gid        => $group,
    managehome => true,
  } ->
  ssh_authorized_key { "key for ${title}":
    ensure => present,
    user   => $title,
    type   => 'rsa',
    key    => $key,
  }
}
```

This code allows you to create user accounts with authorized SSH keys in one resource declaration. This code sample has some notable aspects:

- Since you are essentially wrapping multiple resource types, the titles of all *inner* resources are derived from the instance title (or name) of the current defined type instance; actually, this is a required practice for all defined types
- You can hardcode parts of your business logic; in this example, we dispensed with the support for non-RSA SSH keys and defined `users` as the default group
- Resources inside defined types can and should manage ordering among themselves (using the chaining arrow `->` in this case)

Using defined types as macros

Some source code requires many repetitive tasks. Assume that your site uses a subsystem that relies on symbolic links at a certain location to enable configuration files, just like init does with the symlinks in `rc2.d/` and its siblings, which point back to `./init.d/<service>`.

A manifest that enables a large number of configuration snippets might look like this:

```
file { '/etc/example_app/conf.d.enabled/england':
  ensure => 'link',
  target => '../conf.d.available/england'
}
file { '/etc/example_app/conf.d.enabled/ireland':
  ensure => 'link',
  target => '../conf.d.available/ireland'
}
file { '/etc/example_app/conf.d.enabled/germany':
  ensure => 'link',
  target => '../conf.d.available/germany'
...
}
```

This is tiring to read and somewhat painful to maintain. In a C program, one would use a preprocessor macro that just takes the base name of both link and target and expands to the three lines of each resource description. Puppet does not use a preprocessor, but you can use defined types to achieve a similar result:

```
define example_app_config {
  file { "/etc/example_app/conf.d.enabled/${name}":
    ensure => 'link',
    target => "../conf.d.available/${name}",
  }
}
```



The defined type actually acts more like a simple function call than an actual macro.



The define requires no arguments – it can rely solely on its resource name, so the preceding code can now be simplified to the following:

```
example_app_config {'england': }
example_app_config {'ireland': }
example_app_config {'germany': }
...
```

Alternatively, the following code is even more terse:

```
example_app_config { [ 'england', 'ireland', 'germany', ... ]:
}
```

This array notation leads us to another use of defined types.

Exploiting array values using defined types

One of the more common scenarios in programming is the requirement to accept an array value from some source and perform some task on each value. Puppet manifests are not exempt from this.

Let's assume that the symbolic links from the previous example actually led to directories, and that each such directory would contain a subdirectory to hold optional links to regions. Puppet should manage those links as well.

Of course, after learning about the macro aspect of defined types, you would not want to add each of those regions as distinct resources to your manifest. However, you will need to devise a way to map region names to countries. Seeing as there is already a defined resource type for countries, there is a very direct approach to this: make the list of regions an attribute (or rather, a parameter) of the defined type:

```
define example_app_config (
  Array $regions = []
) {
  file { "/etc/example_app/conf.d.enabled/${name}":
    ensure => link,
    target => "../conf.d.available/${name}",
  }
  # to do: add functionality for $regions
}
```

Using the parameter is straightforward:

```
example_app_config { 'england':
  regions => [ 'South East', 'London' ],
}
example_app_config { 'ireland':
  regions => [ 'Connacht', 'Ulster' ],
}
example_app_config { 'germany':
  regions => [ 'Berlin', 'Bayern', 'Hamburg' ],
}
...
```

The actual challenge is putting these values to use. A naïve approach is to add the following to the definition of `example_app_config`:

```
file { $regions:
  path   => "/etc/example_app/conf.d.enabled/${title}/
    regions/${name}",
  ensure => 'link',
  target => "../../regions.available/${name}";
}
```

However, this will not work. The `$name` variable does not refer to the title of the `file` resource that is being declared. It actually refers, just like `$title`, to the name of the enclosing class or defined type (in this case, the country name). Still, the actual construct will seem quite familiar to you. The only missing piece here is yet another defined type:

```
define example_app_region(String $country) {
  file { "/etc/example_app/conf.d.enabled/${country}/regions/${name}":
    ensure => 'link',
    target => "../../regions.available/${name}",
  }
}
```

The complete definition of the `example_app_config` defined type should look like this then:

```
define example_app_config(Array $regions = []) {
  file { "/etc/example_app/conf.d.enabled/${name}":
    ensure => 'link',
    target => "../conf.d.available/${name}",
  }
  example_app_region { $regions:
    country => $name,
  }
}
```

The *outer* defined type adapts the behavior of the `example_app_region` type to its respective needs by passing its own resource name as a parameter value.

Using iterator functions

With Puppet 4 and later versions, you probably would not write code like the one in the previous section. Thanks to new language features, using defined types as iterators is no longer necessary. We will outline the alternative using the following examples, with a more thorough exploration in *Chapter 7, New Features from Puppet 4*.

The plain country links can now be declared from an Array using the `each` function:

```
[ 'england', 'ireland', 'germany' ].each |$country| {
  file { "/etc/example_app/conf.d.enabled/${country}":
    ensure => 'link',
    target => "../conf.d.available/${country}",
  }
}
```

The regions can be declared from structured data. A hash suffices for this use case:

```
$region_data = {
  'england' => [ 'South East', 'London' ],
  'ireland' => [ 'Connacht', 'Ulster' ],
  'germany' => [ 'Berlin', 'Bayern', 'Hamburg' ],
}
$region_data.each |$country, $region_array| {
  $region_array.each |$region| {
    file { "/etc/example_app/conf.d.enabled/${country}/regions/${region}":
      ensure => link,
      target => "../../regions.available/${region}",
    }
  }
}
```

In new manifests, you should prefer iteration using the `each` and `map` functions over using defined types for this purpose. You will find examples of the former in older manifest code, however. See *Chapter 7, New Features from Puppet 4*, for more information on the topic.

Including classes from defined types

The `example_app_config` type that was defined in the previous example is supposed to serve a very specific purpose. Therefore, it assumes that the base directory, `/etc/example_app`, and its subdirectories were managed independently, outside the defined type. This was a sound design, but many defined types are meant to be used from lots of independent classes or other defined types. Such defines need to be self-contained.

In our example, the defined type needs to make sure that the following resources are part of the manifest:

```
file { [ '/etc/example_app', '/etc/example_app/config.d.enabled' ]:
  ensure => 'directory',
}
```

Just putting this declaration into the body of the define will lead to duplicate resource errors. Each instance of `example_app_config` will try to declare the directories by itself. However, we already discussed a pattern to avoid just that issue – we called it the component class.

To make sure that any instance of the `example_app_config` type is self-contained and works on its own, wrap the preceding declaration in a class (for example, `class example_app_config_directories`) and make sure to include this class right in the body of the define:

```
define example_app_config(Array $regions = []) {
  include example_app_config_directories
  ...
}
```



You can refer to the examples that come with your copy of this book for the definition of the class.

Establishing relationships among containers

Puppet's classes bear little or no similarity to classes that you find in object-oriented programming languages such as Java or Ruby. There are no methods or attributes. There are no distinct instances of any class. You cannot create interfaces or abstract base classes.

One of the few shared characteristics is the encapsulation aspect. Just as classes from OOP, Puppet's classes hide implementation details. To get Puppet to start managing a subsystem, you just need to include the appropriate class.

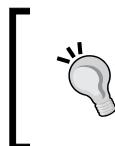
Passing events between classes and defined types

By sorting all resources into classes, you make it unnecessary (for your co-workers or other collaborators) to know about each single resource. This is beneficial. You can think of the collection of classes and defined types as your interface. You would not want to read all manifests that anyone on your project ever wrote.

However, the encapsulation is inconvenient for passing resource events. Say you have some daemon that creates live statistics from your Apache logfiles. It should subscribe to Apache's configuration files so that it can restart if there are any changes (which might be of consequence to this daemon's operation). In another scenario, you might have Puppet manage some external data for a self-compiled Apache module. If Puppet updates such data, you will want to trigger a restart of the Apache service to reload everything.

Armed with the knowledge that there is a service, `Service['apache2']`, defined somewhere in the `apache` class, you can just go ahead and have your module data files notify that resource. It would work—Puppet does not apply any sort of protection to resources that are declared in foreign classes. However, it would pose a minor maintainability issue.

The reference to the resource is located far from the resource itself. When maintaining the manifest later, you or a coworker might wish to look at the resource when encountering the reference. In the case of Apache, it's not difficult to figure out where to look, but in other scenarios, the location of the reference target can be less obvious.



Looking up a targeted resource is usually not necessary, but it can be important to find out what that resource actually does. It gets especially important during debugging, if after a change to the manifest, the referenced resource is no longer found.

Besides, this approach will not work for the other scenario, in which your daemon needs to subscribe to configuration changes. You could blindly subscribe the central `apache2.conf` file, of course. However, this would not yield the desired results if the responsible class opted to do most of the configuration work inside snippets in `/etc/apache2/conf.d`.

Both scenarios can be addressed cleanly and elegantly by directing the `notify` or `subscribe` parameters at the whole class that is managing the entity in question:

```
file { '/var/lib/apache2/sample-module/data01.bin':
  source => '...',
  notify => Class['apache'],
}
service { 'apache-logwatch':
  enable    => true,
  subscribe => Class['apache'],
}
```

Of course, the signals are now sent (or received) indiscriminately—the file not only notifies `Service['apache2']`, but also every other resource in the `apache` class. This is usually acceptable, because most resources ignore events.

As for the `logwatch` daemon, it might refresh itself needlessly if some resource in the `apache` class needs a `sync` action. The odds for this occurrence depend on the implementation of the class. For ideal results, it might be sensible to relocate the configuration file resources into their own class so that the daemon can subscribe to that instead.

With your defined types, you can apply the same rules: subscribe to and notify them as required. Doing so feels quite natural, because they are declared like native resources anyway. This is how you subscribe several instances of the defined type, `symlink`:

```
$active_countries = [ 'England', 'Ireland', 'Germany' ]
service { 'example-app':
  enable    => true,
  subscribe => Symlink[$active_countries],
}
```

Granted, this very example is a bit awkward, because it requires all `symlink` resource titles to be available in an array variable. In this case, it would be more natural to make the defined type instances notify the service instead:

```
symlink { [ 'England', 'Ireland', 'Germany' ]:
  notify => Service['example-app'],
}
```

This notation passes a metaparameter to a defined type. The result is that this parameter value is applied to all resources declared inside the define.

If a defined type wraps or contains a `service` or `exec` type resource, it can also be desirable to notify an instance of that define to refresh the contained resource. The following example assumes that the `service` type is wrapped by a defined type called `protected_service`:

```
file { '/etc/example_app/main.conf':
  source => '....',
  notify => Protected_service['example-app'],
}
```

Ordering containers

The `notify` and `subscribe` metaparameters are not the only ones that you can direct at classes and instances of defined types—the same holds true for their siblings, `before` and `require`. These allow you to define an order for your resources relative to classes, order instances of your defined types, and even order classes among themselves.

The latter works by virtue of the chaining operator:

```
include firewall
include loadbalancing
Class['firewall'] -> Class['loadbalancing']
```

The effect of this code is that all resources from the `firewall` class will be synchronized before any resource from the `loadbalancing` class, and failure of any resource in the former class will prevent all resources in the latter from being synchronized.



Note that the chaining arrow cannot just be placed in between the `include` statements. It works only between resources or references.



Because of these ordering semantics, it is actually quite wholesome to require a whole class. You effectively mark the resource in question as being dependent on the class. As a result, it will only be synchronized if the entire subsystem that the class models is successfully synchronized first.

Limitations

Sadly, there is a rather substantial issue with both the ordering of containers and the distribution of refresh events: both will not transcend the `include` statements of further classes. Consider the following example:

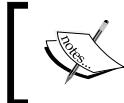
```
class apache {
  include apache::service
  include apache::package
  include apache::config
}
file { '/etc/apache2/conf.d/passwords.conf':
  source  => '...',
  require => Class['apache'],
}
```

I often mentioned how the comprehensive `apache` class models everything about the Apache server subsystem, and in the previous section, I went on to explain that directing a `require` parameter at such a class will make sure that Puppet only touches the dependent resource if the subsystem has been successfully configured.

This is mostly true, but due to the limitation concerning class boundaries, it doesn't achieve the desired effect in this scenario. The dependent configuration file should actually require the `Package['apache']` package, declared in `class apache::package`. However, the relationship does not span multiple class inclusions, so this particular dependency will not be part of the resulting catalog at all.

Similarly, any refresh events sent to the `apache` class will have no effect – they are distributed to resources declared in the class's body (of which there are none), but are not passed on to included classes. Subscribing to the class will make no sense either, because any resource events generated inside the included classes will not be forwarded by the `apache` class.

The bottom line is that relationships to classes cannot be built in utter ignorance of their implementation. If in doubt, you need to make sure that the resources that are of interest are actually declared directly inside the class you are targeting.



The discussion revolved around the example of the `include` statements in classes, but since it is common to use them in defined types as well, the same limitation applies in this case too.

There is a bright side to this as well. A more correct implementation of the Apache configuration file from the example explained would depend on the package, but would also synchronize itself before the service, and perhaps even notify it (so that Apache restarts if necessary). When all resources are part of the apache class and you want to adhere to the pattern of interacting with the container only, it would lead to the following declaration:

```
file { '/etc/apache2/conf.d/passwords.conf':
  source  => '...',
  require => Class['apache'],
  notify   => Class['apache'],
}
```

This forms an instant dependency circle: the `file` resource requires all parts of the `apache` class to be synchronized before it gets processed, but to notify them, they must all be put after the `file` resource in the order graph. This cannot work. With the knowledge of the inner structure of the `apache` class, the user can pick metaparameter values that actually work:

```
file { '/etc/apache2/conf.d/passwords.conf':
  source  => '...',
  require => Class['apache::package'],
  notify   => Class['apache::service'],
}
```

For the curious the preceding code shows what the inner classes look like, roughly.

The other good news is that invoking defined types does not pose the same kind of issue that an `include` statement of a class does. Events are passed to resources inside defined types just fine, transcending an arbitrary number of stacked invocations. Ordering also works just as expected. Let's keep the example brief:

```
class apache {
  virtual_host { 'example.net': ... }
  ...
}
```

This `apache` class also creates a virtual host using the defined type, `virtual_host`. A resource that requires this class will implicitly require all resources from within this `virtual_host` instance. A subscriber to the class will receive events from those resources, and events directed at the class will reach the resources of this `virtual_host`.



There is actually a good reason to make the `include` statements behave differently in this regard. As classes can be included very generously (thanks to their singleton aspect), it is common for classes to build a vast network of includes. By adding a single `include` statement to a manifest, you might unknowingly pull hundreds of classes into this manifest.

Assume, for a moment, that relationships and events transcend this whole network. All manners of unintended effects will be the consequence. Dependency circles will be nearly inevitable. The whole construct will become utterly unmanageable. The cost of such relationships will also grow exponentially. Refer to the next section.

The performance implications of container relationships

There is another aspect that you should keep in mind whenever you are referencing a container type to build a relationship to it. The Puppet agent will have to build a dependency graph from this. This graph contains all resources as nodes and all relationships as edges. Classes and defined types get expanded to all their declared resources. All relationships to the container are expanded to relationships to each resource.

This is mostly harmless, if the other end of the relationship is a native resource. A file that requires a class with five declared resources leads to five dependencies. That does not hurt. It gets more interesting if the same class is required by an instance of a defined type that comprises three resources. Each of these builds a relationship to each of the class's resources, so you end up with 15 edges in the graph.

It gets even more expensive when a container invokes complex defined types, perhaps even recursively.

A more complex graph means more work for the Puppet agent, and its runs will take longer. This is especially annoying when running agents interactively during the debugging or development of your manifest. To avoid the unnecessary effort, consider your relationship declarations carefully, and use them only when they are really appropriate.

Mitigating the limitations

The architects of the Puppet language have devised two alternative approaches to solve the ordering issues. We will consider both, because you might encounter them in existing manifests. In new setups, you should always choose the latter variant.

The anchor pattern

The anchor pattern is the classic workaround for the problem with ordering and signaling in the context of recursive class `include` statements. It can be illustrated by the following example class:

```
class example_app {
  anchor { 'example_app::begin':
    notify => Class['example_app_config'],
  }
  include example_app_config
  anchor { 'example_app::end':
    require => Class['example_app_config'],
  }
}
```

Consider a resource that is placed `before => Class['example_app']`. It ends up in the chain before each anchor, and therefore, also before any resource in `example_app_config`, despite the `include` limitation. This is because the `Anchor['example_app::begin']` pseudo-resource notifies the included class and is therefore ordered before all of its resources. A similar effect works for objects that require the class, by virtue of the `example::end` anchor.

The anchor resource type was created for this express purpose. It is not part of the Puppet core, but has been made available through the `stdlib` module instead (the next chapter will familiarize you with modules). Since it also forwards refresh events, it is even possible to notify and subscribe this anchored class, and events will propagate into and out of the included `example_app_config` class.

The `stdlib` module is available in the Puppet Forge, but more about this in the next chapter. There is a descriptive document for the anchor pattern to be found online as well, in Puppet Labs' Redmine issue tracker (now obsolete) at http://projects.puppetlabs.com/projects/puppet/wiki/Anchor_Pattern. It is somewhat dated, seeing as the anchor pattern has been supplanted as well by Puppet's ability to contain a class in a container.

The contain function

To make composite classes directly work around the limitations of the `include` statement, you can take advantage of the `contain` function found in Puppet version 3.4.x or newer.

If the earlier apache example had been written like the following one, there would have been no issues concerning ordering and refresh events:

```
class apache {
  contain apache::service
  contain apache::package
  contain apache::config
}
```

The official documentation describes the behavior as follows:

"A contained class will not be applied before the containing class is begun, and will be finished before the containing class is finished."

This might read like we're now discussing the panacea for the presented class ordering issues here. Should you just be using `contain` in place of `include` from here on out and never worry about class ordering again? Of course not, this would introduce lots of unnecessary ordering constraints and lead you into unfixable dependency circles very quickly. Do contain classes, but make sure that it makes sense. The contained class should really form a vital part of what the containing class is modeling.



The quoted documentation refers to classes only, but classes can be contained in defined types just as well. The effect of containment is not limited to ordering aspects either. Refresh events are also correctly propagated.

Making classes more flexible through parameters

Up until this point, classes and defines were presented as direct opposites with respect to flexibility; defined types are inherently adaptable through different parameter values, whereas classes model just one static piece of state. As the section title suggests, this is not entirely true. Classes too can have parameters. Their definition and declaration become rather similar to those of defined types in this case:

```
class apache::config(Integer $max_clients=100) {
  file { '/etc/apache2/conf.d/max_clients.conf':
```

```
    content => "MaxClients ${max_clients}\n",
}
}
```

With a definition like the preceding one, the class can be declared with a parameter value:

```
class { 'apache::config':
  max_clients => 120,
}
```

This enables some very elegant designs, but introduces some drawbacks as well.

The caveats of parameterized classes

The consequence of allowing class parameters is almost obvious: you lose the singleton characteristic. Well, that's not entirely true either, but your freedom in declaring the class gets limited drastically.

Classes that define default values for all parameters can still be declared with the `include` statement. This can still be done an arbitrary number of times in the same manifest.

However, the resource-like declaration of `class { 'name' : }` cannot appear more than once for any given class in the same manifest. This is in keeping with the rules for resources and should not be very surprising – after all, it would be very awkward to try and bind different values to a class's parameters in different locations throughout the manifest.

Things become very confusing when mixing `include` with the alternative syntax though. It is valid to include a class an arbitrary number of times after it has been declared using the resource-like notation. However, you cannot use the resource style declaration *after* a class has been declared using `include`. That's because the parameters are then determined to assume their default values, and a `class { 'name' : }` declaration clashes with that.

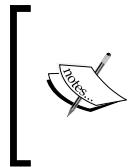
In a nutshell, the following code works:

```
class { 'apache::config': }
include apache::config
```

However, the following code does not work:

```
include apache::config
class { 'apache::config': }
```

As a consequence, you effectively cannot add parameters to component classes, because the `include` statement is no longer safe to use in large quantities. Therefore, parameters are essentially only useful for comprehensive classes, which usually don't get included from different parts of the manifest.



In *Chapter 5, Extending Your Puppet Infrastructure with Modules*, we will discuss some alternate patterns, some of which exploit class parameters. Also note that *Chapter 8, Separating Data from Code Using Hiera*, presents a solution that gives you more flexibility with parameterized classes. Using this, you can be more liberal with your class interfaces.

Preferring the `include` keyword

Ever since class parameters have been available, some Puppet users have felt compelled to write (example) code that would make it a point to forgo the `include` keyword in favor of resource-like class declarations, such as this:

```
class apache {
  class { 'apache::service': }
  class { 'apache::package': }
  class { 'apache::config': }
}
```

Doing this is a very bad idea. We cannot stress this enough: one of the most powerful concepts about Puppet's classes is their singleton aspect—the ability to include a class in a manifest arbitrarily and without worrying about clashes with other code. The mentioned declaration syntax deprives you of this power, even when the classes in question don't support parameters.

The safest route is to use `include` whenever possible, and to avoid the alternate syntax whenever you can. In fact, *Chapter 8, Separating Data from Code Using Hiera*, introduces the ability to use class parameters without the resource-like class declaration. This way, you can rely solely on `include`, even when parameters are in play. These are the safest recommended practices to keep you out of trouble from incompatible class declarations.

Summary

Classes and defined types are the essential tools to create reusable Puppet code. While classes hold resources that must not be repeated in a manifest, the define is capable of managing a distinct set of adapted resources upon every invocation. It does that by leveraging the parameter values it receives. While classes do support parameters as well, there are some caveats to bear in mind.

To use defined types in your manifest, you declare instances just like resources of native types. Classes are mainly used through the `include` statement, although there are alternatives such as the `class { }` syntax or the `contain` function.

There are also some ordering issues with classes that the `contain` function can help mitigate.

In theory, classes and defines suffice to build almost all the manifests that you will ever need. In practice, you will want to organize your code into larger structures. The next chapter will show you how to do exactly that, and introduce you to a whole range of useful functionality beyond it.

5

Extending Your Puppet Infrastructure with Modules

In the previous chapter, you learned about the tools that create modularized and reusable Puppet code in the form of classes and defined types. We discussed that almost all the Puppet resources should be separated into appropriate classes, except if they logically need to be part of a defined type. This is almost enough syntax to build manifests for an entire fleet of agent nodes - each selecting the appropriate composite classes, which in turn include further required classes, with all the classes recursively instantiating the defined types.

What has not been discussed up until now is the organization of the manifests in the filesystem. It is obviously undesirable to stuff all of your code into one large `site.pp` file. The answer to this problem is provided by modules and will be explained in this chapter.

Besides organizing classes and defines, modules are also a way to share common code. They are software libraries for Puppet manifests and plugins. They also offer a convenient place to locate the interface descriptions that were hinted at in the previous chapter. Puppet Labs runs a dedicated service for hosting open source modules, called the Puppet Forge.

The existence and general location of the modules were mentioned briefly in *Chapter 3, A Peek under the Hood – Facts, Types, and Providers*. It is now time to explore these and other aspects in greater detail. We'll cover the following topics in this chapter:

- An overview of Puppet's modules
- Maintaining environments
- Following modules' best practices
- Building a specific module
- Finding helpful Forge modules

An overview of Puppet's modules

A module can be seen as a higher-order organizational unit. It bundles up classes and defined types that contribute to a common management goal (specific system aspects or a piece of software, for example). These manifests are not all that is organized through modules; most modules also bundle files and file templates. There can also be several kinds of Puppet plugins in a module. This section will explain these different parts of a module and show you where they are located. You will also learn about the means of module documentation and how to obtain existing modules for your own use.

Parts of a module

For most modules, **manifests** form the most important part - the core functionality. The manifests consist of classes and defined types, which all share a namespace, rooted at the module name. For example, an `ntp` module will contain only classes and defines whose names start with the `ntp::` prefix.

Many modules contain files that can be synced to the agent's filesystem. This is often used for configuration files or snippets. You have seen examples of this, but let's repeat them. A frequent occurrence in many manifests is `file` resources such as the following:

```
file { '/etc/ntp.conf':
  source => 'puppet:///modules/ntp/ntp.conf',
}
```

The above resource references a file that ships with a hypothetical `ntp` module. It has been prepared to provide generally suitable configuration data. However, there is often a need to tweak some parameters inside such a file, so that the node manifests can declare customized config settings for the respective agent. The tool of choice for this is templates, which will be discussed in the next chapter.

Another possible component of a module that you have already read about is Custom Facts – code that gets synchronized to the agent and runs before a catalog is requested, so that the output becomes available as facts about the agent system.

These facts are not the only Puppet plugins that can be shipped with modules. There are also **parser functions** (also called **custom functions**), for one. These are actual functions that you can use in your manifests. In many situations, they are the most convenient way, if not the only way, to build some specific implementations.

The final plugin type that has also been hinted at in an earlier chapter is the custom native types and providers, which are conveniently placed in modules as well.

Module structure

All the mentioned components need to be located in specific filesystem locations for the master to pick them up. Each module forms a directory tree. Its root is named after the module itself. For example, the `ntp` module is stored in a directory called `ntp/`.

All manifests are stored in a subdirectory called `manifests/`. Each class and defined type has its own respective file. The `ntp::package` class will be found in `manifests/package.pp`, and the defined type called `ntp::monitoring::nagios` will be found in `manifests/monitoring/nagios.pp`. The first particle of the container name (`ntp`) is always the module name, and the rest describes the location under `manifests/`. You can refer to the module tree in the following paragraphs for more examples.

The `manifests/init.pp` file is special. It can be thought of as a default manifest location, because it is looked up for any definition from the module in question. Both the examples that were just mentioned can be put into `init.pp` and will still work. Doing this makes it harder to locate the definitions, though.

In practice, `init.pp` should only hold one class, which is named after the module (such as the `ntp` class), if your module implements such a class. This is a common practice, as it allows the manifests to use a simple statement to tap the core functionality of the module:

```
include ntp
```

You can refer to the *Modules' best practices* section for some more notes on this subject.

The files and templates that a module serves to the agents are not strictly sorted into specific locations. It is only important that they be placed in the `files/` and `templates/` subdirectories, respectively. The contents of these subtrees can be structured to the module author's liking, and the manifest must reference them correctly. Static files should always be addressed through URLs, such as these:

```
puppet:///modules/ntp/ntp.conf  
puppet:///modules/my_app/opt/scripts/find_my_app.sh
```

These files are found in the corresponding subdirectories of `files/`:

```
.../modules/ntp/files/ntp.conf  
.../modules/my_app/files/opt/scripts/find_my_app.sh
```

The `modules` prefix in the URI is mandatory and is always followed by the module name. The rest of the path translates directly to the contents of the `files/` directory. There are similar rules for templates. You can refer to *Chapter 6, Leveraging the Full Toolset of the Language*, for the details.

Finally, all plugins are located in the `lib/` subtree. Custom facts are Ruby files in `lib/facter/`. Parser functions are stored in `lib/puppet/parser/functions/`, and for custom resource types and providers, there is `lib/puppet/type/` and `lib/puppet/provider/`, respectively. This is not a coincidence; these Ruby libraries are looked up by the master and the agent in the according namespaces. There are examples for all these components later in this chapter.

In short, following are the contents of a possible module in a tree view:

```
/opt/puppetlabs/code/environments/production/modules/my_app
  templates      # templates are covered in the next chapter
  files
    subdir1      # puppet:///modules/my_app/subdir1/<filename>
    subdir2      # puppet:///modules/my_app/subdir2/<filename>
    subsubdir    # puppet:///modules/my_app/subdir2/subsubdir/...
  manifests
    init.pp       # class my_app           is defined here
    params.pp     # class my_app::params   is defined here
    config
      detail.pp  # my_app::config::detail  is defined here
      basics.pp  # my_app::config::basics  is defined here
  lib
    facter        # contains .rb files with custom facts
    puppet
      functions   # contains .rb files with Puppet 4 functions
      parser
        functions # contains .rb files with parser functions
    type          # contains .rb files with custom types
    provider      # contains .rb files with custom providers
```

Documentation in modules

A module can and should include documentation. The Puppet master does not process any module documentation by itself. As such, it is largely up to the authors to decide how to structure the documentation of the modules that are created for their specific site only. That being said, there are some common practices and it's a good idea to adhere to them. Besides, if a module should end up being published on the Forge, appropriate documentation should be considered mandatory.



The process of publishing modules is beyond the scope of this book.
You can find a guide at https://docs.puppetlabs.com/puppet/latest/reference/modules_publishing.html.

For many modules, the main focus of the documentation is centered on the `README` file, which is located right in the module's root directory. It is customarily formatted in Markdown as `README.md` or `README.markdown`. The `README` file should contain explanations, and often, there is a reference documentation as well.

Puppet DSL interfaces can also be documented right in the manifest, in the `rdoc` and `YARD` format. This applies to classes and defined types:

```
# Class: my_app::firewall
#
# This class adds firewall rules to allow access to my_app.
#
# Parameters: none
class my_app::firewall {
    # class code here
}
```

You can generate HTML documentation (including navigation) for all your modules using the `puppet doc` subcommand. This practice is somewhat obscure, so it won't be discussed here in great detail. However, if this option is attractive to you, we encourage you to peruse the documentation.

The following command is a good starting point:

```
puppet help doc
```

Another useful resource is the `puppetlabs-strings` module (<https://forge.puppetlabs.com/puppetlabs/strings>), which will eventually supersede `puppet doc`.

Plugins are documented right in their Ruby code. There are examples for this in the following sections.

Maintaining environments

Puppet doesn't organize things in modules exclusively. There is a higher-level unit called **environment** that groups and contains the modules. An environment mainly consists of:

- One or more site manifest files
- A `modules` directory
- An optional `environment.conf` configuration file

When the master compiles the manifest for a node, it uses exactly one environment for this task. As described in *Chapter 2, The Master and Its Agents*, it always starts in `manifests/*.pp`, which form the environment's site manifest. Before we take a look at how this works in practice, let's see an example environment directory:

```
/opt/puppetlabs/code/environments
  production
    environment.conf
    manifests
      site.pp
      nodes.pp
    modules
      my_app
      ntp
```

The `environment.conf` file can customize the environment. Normally, Puppet uses `site.pp` and the other files in the `manifests` directory. To make Puppet read all the `pp` files in another directory, set the `manifest` option in `environment.conf`:

```
# /opt/puppetlabs/code/environments/production/environment.conf
manifest = puppet_manifests
```

In most circumstances, the `manifest` option need not be changed.

The `site.pp` file will include classes and instantiate defines from the modules. Puppet looks for modules in the `modules` subdirectory of the active environment. You can define additional subdirectories that hold the modules by setting the `modulepath` option in `environment.conf`:

```
# /opt/puppetlabs/code/environments/production/environment.conf
modulepath = modules:site-modules
```

The directory structure can be made more distinctive:

```
/opt/puppetlabs/code/environments/
  production
    manifests
    modules
      ntp
    site-modules
      my_app
```

Configuring environment locations

Puppet uses the production environment by default. This and the other environments are expected to be located in `/opt/puppetlabs/code/environments`. You can override this default by setting the `environmentpath` option in `puppet.conf`:

```
[main]
environmentpath = /etc/local/puppet/environments
```

With Puppet 3, you are required to set this option, as it is not yet the default. It is available from version 3.5. The earlier versions needed to configure the environments right in `puppet.conf` with their respective `manifest` and `modulepath` settings. These work just like the settings from `environment.conf`:

```
# in puppet.conf (obsolete since 4.0)
[testing]
manifest = /etc/puppet/environments/testing/manifests
modulepath = /etc/puppet/environments/testing/modules
```

For the special production environment, these Puppet setups use the `manifest` and `modulepath` settings from the `[main]` or `[master]` section. The old default configuration had the production environment look for the manifests and the modules right in `/etc/puppet`:

```
# Obsolete! Works only with Puppet 3.x!
[main]
manifest = /etc/puppet/site.pp
modulepath = /etc/puppet/modules
```

The sites that operate like this today should be migrated to the aforementioned directory environments in `/opt/puppetlabs/code/environments` or similar locations.

Obtaining and installing modules

Downloading existing modules is very common. Puppet Labs hosts a dedicated site for sharing and obtaining the modules - the Puppet Forge. It works just like RubyGems or CPAN and makes it simple for the user to retrieve a given module through a command-line interface. In the Forge, the modules are fully named by prefixing the actual module name with the author's name, such as `puppetlabs-stdlib` or `ffrank-constraints`.

The `puppet module install` command installs a module in the active environment:

```
root@puppetmaster# puppet module install puppetlabs-stdlib
```



The *Testing your modules* section has information on using different environments.



The current release of the `stdlib` module (authored by the user `puppetlabs`) is downloaded from the Forge and installed in the standard modules' location. This is the first location in the current environment's `modulepath`, which is usually the `modules` subdirectory. Specifically, the modules will most likely end up in the `environments/production/modules` directory.



The `stdlib` module in particular should be considered mandatory; it adds a large number of useful functions to the Puppet language. Examples include the `keys`, `values`, and `has_key` functions, which are essential for implementing the proper handling of hash structures, to name only a few. The functions are available to your manifests as soon as the module is installed - there is no need to include any class or other explicit loading. If you write your own modules that add functions, these are loaded automatically in the same way.



Modules' best practices

With all the current versions of Puppet, you should make it a habit to put all the manifest code into modules, with only the following few exceptions:

- The node blocks
- The `include` statements for very select classes that should be omnipresent (the most common design pattern does this in the so-called base role, however; see *Chapter 8, Separating Data from Code Using Hiera*, for the Roles and Profiles pattern)
- Declarations of helpful variables that should have the same availability as the Facter facts in your manifests

This section provides details on how to organize your manifests accordingly. It also advises some design practices and strategies in order to test the changes to the modules.

Putting everything in modules

You might find some manifests in very old installations that gather lots of manifest files in one or more directories and use the `import` statements in the `site.pp` file, such as:

```
import '/etc/puppet/manifests/custom/*.pp'
```

All classes and defined types in these files are then available globally.



This whole approach had scalability issues and has long been deprecated. The `import` keyword is missing from Puppet 4 and the newer versions.



It is far more efficient to give meaningful names to the classes and defined types so that Puppet can look them up in the collection of modules. The scheme has been discussed in an earlier section already, so let's just look at another example where the Puppet compiler encounters a class name, such as:

```
include ntp::server::component::watchdog
```

Puppet will go ahead and locate the `ntp` module in all the configured module locations of the active environment (path names in the `modulepath` setting). It will then try and read the `ntp/manifests/server/component/watchdog.pp` file in order to find the class definition. Failing this, it will try `ntp/manifests/init.pp`.

This makes compilation very efficient. Puppet dynamically identifies the required manifest files and includes only those for parsing. It also aids code inspection and development, as it is abundantly clear where you should look for specific definitions.



Technically, it is possible to stuff all of a module's manifests into its `init.pp` file, but you lose the advantages that a structured tree of module manifests offers.



Avoiding generalization

Each module should ideally serve a specific purpose. On a site that relies on Puppet to manage a diverse server infrastructure, there are likely modules for each respective service, such as `apache`, `ssh`, `nagios`, `nginx`, and so forth. There can also be site-specific modules such as `users` or `shell_settings` if the operations require this kind of fine-grained control. Such customized modules are sometimes just named after the group or the company that owns them.

The ideal granularity depends on the individual requirements of your setup. What you generally want to avoid are modules with names such as `utilities` or `helpers`, which serve as a melting pot for ideas that don't fit in any of the existing modules. Such a lack of organization can be detrimental to discipline and can lead to chaotic modules that include definitions that should have become their own respective modules instead.

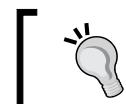
Adding more modules is cheap. A module generally incurs no cost for the Puppet master operation, and your user experience will usually become more efficient with more modules, not less so. Of course, this balance can tip if your site imposes special documentation or other handling prerequisites on each module. Such rulings must then be weighed into the decisions about module organization.

Testing your modules

Depending on the size of your agent network, some or many of your modules can be used by a large variety of nodes. Despite these commonalities, these nodes can be quite different from one another. A change to a central module, such as `ssh` or `ntp`, which are likely used by a large number of agents, can have quite extensive consequences.

The first and the most important tool for testing your work is the `--noop` option for Puppet. It works for `puppet agent`, as well as `puppet apply`. If it is given on the command line, Puppet will not perform any necessary sync actions, and merely present the respective line of output to you instead. There is an example of this in *Chapter 1, Writing Your First Manifests*.

When using a master instead of working locally with `puppet apply`, a new problem arises, though. The master is queried by all your agents. Unless all the agents are disabled while you are testing a new manifest, it is very likely that one will check in and accidentally run the untested code.



In fact, even your test agent can trigger its regular run while you are logged in, transparently in the background.

It is very important to guard against such uncontrolled manifest applications. A small mistake can damage a number of agent machines in a short time period. The best way to go about this is to define multiple environments on the master.

Safe testing with environments

Besides the production environment, you should create at least one testing environment. You can call it `testing` or whatever you like. When using the directory environments, just create its directory in `environmentpath`.

Such an additional environment is very useful for testing changes. The test environment or environments should be copies of the production data. Prepare all the manifest changes in `testing` first. You can make your agents test this change before you copy it to production:

```
root@agent# puppet agent --test --noop --env testing
```

You can even omit the `noop` flag on some or all of your agents so that the change is actually deployed. Some subtle mistakes in the manifests cannot be detected from an inspection of the `noop` output, so it is usually a good idea to run the code at least once before releasing it.



Environments are even more effective when used in conjunction with source control, especially distributed systems such as `git` or `mercurial`. Versioning your Puppet code is a good idea independently of environments and testing – this is one of the greatest advantages that Puppet has to offer you through its **Infrastructure as Code** paradigm.

Using environments and the `noop` mode form a pragmatic approach to testing that can serve in most scenarios. The safety against erroneous Puppet behavior is limited, of course. There are more formal ways of testing the modules:

- The `rspec-puppet` tool allows the module authors to implement unit tests based on `rspec`. You can find more details at <http://rspec-puppet.com/>.
- Acceptance testing can be performed through `beaker`. You can refer to <https://github.com/puppetlabs/beaker/wiki/How-To-Beaker> for details.

Explaining these tools in detail is beyond the scope of this book.

Building a specific module

This chapter has discussed many theoretical and operational aspects of modules, but you are yet to gain an insight into the process of writing modules. For this purpose, the rest of this chapter will have you create an example module step by step.

It should be stressed again that for the most part, you will want to find general purpose modules from the Forge. The number of available modules is ever growing, so the odds are good that there is something already there to help you with what you need to do.

Assume that you want to add Cacti to your network, an RRD tool-based trend monitor and graphing server, including a web interface. If you would check the Forge first, you would indeed find some modules. However, let's further assume that none of them speak to you, because either the feature set or the implementation is not to your liking. If even the respective interfaces don't meet your requirements, it doesn't make much sense to base your own module on an existing one (in the form of a fork on GitHub) either. You will then need to write your own module from scratch.

Naming your module

Module names should be concise and to the point. If you manage a specific piece of software, name your module after it - apache, java, mysql, and so forth. Avoid verbs such as `install_cacti` or `manage_cacti`. If your module name does need to consist of several words (because the target subsystem has a long name), they should be divided by underscore characters. Spaces, dashes, and other non-alphanumeric characters are not allowed.

In our example, the module should just be named `cacti`.

Making your module available to Puppet

To use your own module, you don't need to make it available for installation through `puppet module`. For that, you will need to upload the module to the Forge first, which will require quite some additional effort. Luckily, a module will work just fine without all this preparation, if you just put the source code in the proper location on your master.

To create your own `cacti` module, create the basic directories:

```
root@puppetmaster# mkdir -p /opt/puppetlabs/code/environments/testing/
cacti/{manifests,files}
```

Don't forget to synchronize all the changes to production once the agents use them.

Implementing basic module functionality

Most modules perform all of their work through their manifests.



There are notable exceptions, such as the `stdlib` module. It mainly adds the parser functions and a few general-purpose resource types.

When planning the classes for your module, it is most straightforward to think about how you would like to use the finished module. There is a wide range of possible interface designs. The de facto standard stipulates that the managed subsystem is initialized on the agent system by including the module's main class - the class that bears the same name as the module and is implemented in the module's `init.pp` file.

For our Cacti module, the user should use the following:

```
include cacti
```

As a result, Puppet would take all the required steps in order to install the software and if necessary, perform any additional initialization.

Start by creating the `cacti` class and implementing the setup in the way you would from the command line, replacing the commands with appropriate Puppet resources. On a Debian system, installing the `cacti` package is enough. Other required software is brought in through the dependencies (completing the LAMP stack), and after the package installation, the interface becomes available through the web URI `/cacti/` on the server machine:

```
# .../modules/cacti/manifests/init.pp
class cacti {
  package { 'cacti':
    ensure => installed,
  }
}
```

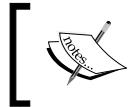
Your module is now ready for testing. Invoke it from your agent's manifest in `site.pp` or `nodes.pp` of the testing environment:

```
node 'agent' {
  include cacti
}
```

Apply it on your agent directly:

```
root@agent# puppet agent --test --environment testing
```

This will work on Debian, and Cacti is reachable via `http://<address>/cacti/`.



Some sites use an External Node Classifier (ENC), such as The Foreman. Among other helpful things, it can centrally assign environments to the nodes. In this scenario, the `--environment` switch will not work.

It's unfortunate that the Cacti web interface will not come up when the home page is requested through the `/` URI. To enable this, give the module the ability to configure an appropriate redirection. Prepare an Apache configuration snippet in the module in `/opt/puppetlabs/code/environments/testing/cacti/files/etc/apache2/conf.d/cacti-redirect.conf`:

```
# Do not edit this file - it is managed by Puppet!
RedirectMatch permanent ^/$ /cacti/
```



The warning notice is helpful, especially when multiple administrators have access to the Cacti server.

It makes sense to add a dedicated class that will sync this file to the agent machine:

```
# .../modules/cacti/manifests/redirect.pp
class cacti::redirect {
    file { '/etc/apache2/conf.d/cacti-redirect.conf':
        ensure  => file,
        source  => 'puppet:///modules/cacti/etc/apache2/conf.d/
                    cacti-redirect.conf',
        require => Package['cacti'];
    }
}
```



A short file like this can also be managed through the `file` type's `content` property instead of `source`:

```
$puppet_warning = '# Do not edit - managed by Puppet!'
$line = 'RedirectMatch permanent ^/$ /cacti/'
file { '/etc/apache2/conf.d/cacti-redirect.conf':
    ensure  => file,
    content => "${puppet_warning}\n${line}\n",
}
```

This is more efficient, because the content is part of the catalog and so the agent does not need to retrieve the checksum through another request to the master.

The module now allows the user to include `cacti::redirect` in order to get this functionality. This is not a bad interface as such, but this kind of modification is actually well-suited to become a parameter of the `cacti` class:

```
class cacti($redirect = true) {
  if $redirect {
    contain cacti::redirect
  }
  package { 'cacti':
    ensure => installed,
  }
}
```

The redirect is now installed by default when a manifest uses `include cacti`. If the web server has other virtual hosts that serve things that are not Cacti, this might be undesirable. In such cases, the manifest will declare the class with this following parameter:

```
class { 'cacti': redirect => false }
```

Speaking of best practices, most modules will also separate the installation routine into a class of its own. In our case, this is hardly helpful, because the installation status is ensured through a single resource, but let's do it anyway:

```
class cacti( $redirect = true ) {
  contain cacti::install
  if $redirect {
    contain cacti::redirect
  }
}
```

It's sensible to use `contain` here in order to make the Cacti management a solid unit. The `cacti::install` class is put into a separate `install.pp` manifest file:

```
# .../modules/cacti/manifests/install.pp
class cacti::install {
  package { 'cacti':
    ensure => 'installed'
  }
}
```

On Debian, the installation process of the `cacti` package copies another Apache configuration file to `/etc/apache2/conf.d`. Since Puppet performs a normal `apt` installation, this result will be achieved. However, Puppet does not make sure that the configuration *stays* in this desired state.

 There is an actual risk that the configuration might get broken. If the `puppetlabs-apache` module is in use for a given node, it will usually purge any unmanaged configuration files from the `/etc/apache2/` tree. Be very careful when you enable this module for an existing server. Test it in the `noop` mode. If required, amend the manifest to include the existing configuration.

It is prudent to add a `file` resource to the manifest that keeps the configuration snippet in its post-installation state. Usually with Puppet, this will require you to copy the config file contents to the module, just like the redirect configuration is in a file on the master. However, since the Debian package for Cacti includes a copy of the snippet in `/usr/share/doc/cacti/cacti.apache.conf`, you can instruct the agent to sync the actual configuration with that. Perform this in yet another de facto standard for modules - the `config` class:

```
# .../modules/cacti/manifests/config.pp
class cacti::config {
  file { '/etc/apache2/conf.d/cacti.conf':
    mode   => '0644',
    source => '/usr/share/doc/cacti/cacti.apache.conf'
  }
}
```

This class should be contained by the `cacti` class as well. Running the agent again will now have no effect, because the configuration is already in place.

Creating utilities for derived manifests

You have now created several classes that compartmentalize the basic installation and configuration work for your module. Classes lend themselves very well to implement global settings that are relevant for the managed software as a whole.

However, just installing Cacti and making its web interface available is not an especially powerful capability - after all, the module does little beyond what a user can achieve by installing Cacti through the package manager. The much greater pain point with Cacti is that it usually requires configuration via its web interface; adding servers as well as choosing and configuring graphs for each server can be an arduous task and require dozens of clicks per server, depending on the complexity of your graphing needs.

This is where Puppet can be the most helpful. A textual representation of the desired states allows for quick copy-and-paste repetition and name substitution through regular expressions. Better yet, once there is a Puppet interface, the users can devise their own defined types in order to save themselves from the copy and paste work.

Speaking of defined types, they are what is required for your module to allow this kind of configuration. Each machine in Cacti's configuration should be an instance of a defined type. The graphs can have their own type as well.

As with the implementation of the classes, the first thing you always need to ask yourself is how this task would be done from the command line.



Actually, the better question can be what API you should use for this, preferably from Ruby. However, this is only important if you intend to write Puppet plugins - resource types and providers. We will look into this later in this very chapter.

Cacti comes with a set of CLI scripts. The Debian package makes these available in `/usr/share/cacti/cli`. Let's discover these while we step through the implementation of the Puppet interface. The goals are defined types that will effectively wrap the command-line tools so that Puppet can always maintain the defined configuration state through appropriate queries and update commands.

Adding configuration items

While designing more capabilities for the Cacti module, first comes the ability to register a machine for monitoring - or rather, a **device**, as Cacti itself calls it (network infrastructure such as switches and routers are frequently monitored as well, and not only computers). The name for the first defined type should, therefore, be `cacti::device`.



The same warnings from the *Naming your module* subsection apply - don't give in to the temptation of giving names such as `create_device` or `define_domain` to your type, unless you have very good reasons, such as the removal being impossible. Even then, it's probably better to skip the verb.

The CLI script used to register a device is named `add_device.php`. Its help output readily indicates that it requires two parameters, which are `description` and `ip`. A custom description of an entity is often a good use for the respective Puppet resource's title. The type almost writes itself now:

```
# .../modules/cacti/manifests/device.pp
define cacti::device ($ip) {
  $cli = '/usr/share/cacti/cli'
  $options = "--description='${name}' --ip='${ip}'"
  exec { "add-cacti-device-${name}":
    command => "${cli}/add device.php ${options}",
    require => Class['cacti'],
  }
}
```



In practice, it is often unnecessary to use so many variables, but it serves readability with the limited horizontal space of the page.

This `exec` resource gives Puppet the ability to use the CLI to create a new device in the Cacti configuration. Since PHP is among the Cacti package's requirements, it's sufficient to make the `exec` resource require the `cacti` class. Note the use of `$name`, not only for the `--description` parameter but in the resource name for the `exec` resource as well. This ensures that each `cacti::device` instance declares a unique `exec` resource in order to create itself.

However, this still lacks an important aspect. Written as in the preceding example, this `exec` resource will make the Puppet agent run the CLI script always, under any circumstances. This is incorrect though - it should only run if the device has not yet been added.

Every `exec` resource should have one of the `creates`, `onlyif`, or `unless` parameters. It defines a query for Puppet to determine the current sync state. The `add_device` call must be made *unless* the device exists already. The query for the existing devices must be made through the `add_graphs.php` script (counterintuitively). When called with the `--list-hosts` option, it prints one header line and a table of devices, with the description in the fourth column. The following `unless` query will find the resource in question:

```
$search = "sed 1d | cut -f4- | grep -q '^${name}\$'"  
exec { "add-cacti-device-${name}":  
    command => "${cli}/add_device.php ${options}",  
    path    => '/bin:/usr/bin',  
    unless   => "${cli}/add_graphs.php --list-hosts | ${search}",  
    require  => Class[cacti],  
}
```

The `path` parameter is useful as it allows for calling the core utilities without the respective full path.



It is a good idea to generally set a standard list of search paths, because some tools will not work with an empty PATH environment variable.

The `unless` command will return 0 if the exact resource title is found among the list of devices. The final \$ sign is escaped so that Puppet includes it in the `$search` command string literally.

You can now test your new define by adding the following resource to the agent machine's manifest:

```
# in manifests/nodes.pp
node 'agent' {
  include cacti
  cacti::device { 'Puppet test agent (Debian 7)':
    ip => $ipaddress,
  }
}
```

On the next `puppet agent --test` run, you will be notified that the command for adding the device has been run. Repeat the run, and Puppet will determine that everything is now already synchronized with the catalog.

Allowing customization

The `add_device.php` script has a range of optional parameters that allow the user to customize the device. The Puppet module should expose these dials as well. Let's pick one and implement it in the `cacti::device` type. Each Cacti device has a `ping_method` that defaults to `tcp`. With the module, we can even superimpose our own defaults over those of the software:

```
define cacti::device(
  $ip,
  $ping_method='icmp')
{
  $cli = '/usr/share/cacti/cli'
  $base_opt = "--description='${name}' --ip='${ip}'"
  $ping_opt = "--ping_method=${ping_method}"
  $options = "${base_opt} ${ping_opt}"
  $search = "sed 1d | cut -f4- | grep -q '^${name}\$'"
  exec { "add-cacti-device-${name}":
    command => "${cli}/add_device.php ${options}",
    path    => '/bin:/usr/bin',
    unless   => "${cli}/add_graphs.php --list-hosts | ${search}",
    require  => Class[cacti],
  }
}
```

The module uses a default of `icmp` instead of `tcp`. The value is always passed to the CLI script, whether it was passed to the `cacti::device` instance or not. The parameter default is used in the latter case.

 If you plan to publish your module, it is more sensible to try and use the same defaults as the managed software whenever possible.

Once you incorporate all the available CLI switches, you would have successfully created a Puppet API in order to add devices to your Cacti configuration, giving the user the benefits of easy reproduction, sharing, implicit documentation, simple versioning, and more.

Removing unwanted configuration items

There is still one remaining wrinkle. It is atypical for Puppet types to be unable to remove the entities that they create. As it stands, this is a technical limitation of the CLI that powers your module, because it does not implement a `remove_device` function yet. Such scripts have been made available on the Internet, but are not properly a part of Cacti at the time of writing this.

To give the module more functionality, it would make sense to incorporate additional CLI scripts among the module's files. Put the appropriate file into the right directory under `modules/cacti/files/` and add another `file` resource to the `cacti::install` class:

```
file { '/usr/share/cacti/cli/remove_device.php':
  mode    => 755,
  source  =>
    'puppet:///modules/cacti/usr/share/cacti/cli/
remove_device.php',
  require => Package['cacti'],
}
```

You can then add an `ensure` attribute to the `cacti::device` type:

```
define cacti::device(
  $ensure='present',
  $ip,
  $ping_method='icmp')
{
  $cli = '/usr/share/cacti/cli'
  $search = "sed 1d | cut -f4- | grep -q '^${name}\$'"
  case $ensure {
    'present': {
      # existing cacti::device code goes here
    }
  }
}
```

```

        }
      'absent': {
        $remove = "${cli}/remove_device.php"
        $get_id = "$remove --list-devices | awk -F'\\t'
          '\$4==\"${name}\" { print \$1 }'"
        exec { "remove-cacti-device-${name}":
          command => "$remove --device-id=\$( ${get_id} )",
          path    => '/bin:/usr/bin',
          onlyif  => "${cli}/add_graphs.php --list-hosts |
            ${search}",
          require => Class[cacti],
        }
      }
    }
}

```

Note that we took some liberties with the indentation here so as to not break too many lines. This new `exec` resource is quite a mouthful, because the `remove_device.php` script requires the numeric ID of the device to be removed. This is retrieved with a `--list-devices` call that is piped to `awk`. To impair readability even more, some things such as double quotes, `$` signs, and backslashes must be escaped so that Puppet includes a valid `awk` script in the catalog.

Also note that the query for the sync state of this `exec` resource is identical to the one for the `add` resource, except that now it is used with the `onlyif` parameter: *only* take action *if* the device in question is still found in the configuration.

Dealing with complexity

The commands we implemented for the `cacti::device` define are quite convoluted. At this level of complexity, shell one-liners become unwieldy for powering Puppet's resources. It gets even worse when handling the Cacti graphs; the `add_graphs.php` CLI script requires numeric IDs of not only the devices, but of the graphs as well. At this point, it makes sense to move the complexity out of the manifest and write wrapper scripts for the actual CLI. I will just sketch the implementation. The wrapper script will follow this general pattern.

```

#!/bin/bash
DEVICE_DESCR=$1
GRAPH_DESCR=$2
DEVICE_ID=` #scriptlet to retrieve numeric device ID`
GRAPH_ID=` #scriptlet to retrieve numeric graph ID`
GRAPH_TYPE=`#scriptlet to determine the graph type`
/usr/share/cacti/cli/add_graphs.php \
  --graph-type=$GRAPH_TYPE \
  --graph-template-id=$GRAPH_ID \
  --host-id=$DEVICE_ID

```

With this, you can add a straightforward graph type:

```
define cacti::graph($device,$graph=$name) {
    $add = '/usr/local/bin/cacti-add-graph'
    $find = '/usr/local/bin/cacti-find-graph'
    exec { "add-graph-$name-to-$device":
        command => "${add} '$device' '${graph}'",
        path     => '/bin:/usr/bin',
        unless   => "${find} '$device' '${graph}'",
    }
}
```

This also requires an additional `cacti-find-graph` script. Adding this poses an additional challenge as the current CLI has no capabilities for listing configured graphs. There are many more functionalities that can be added to a `cacti` module, such as the management of Cacti's data sources and the ability to change options of the devices and, possibly, other objects that already exist in the configuration.

Such commodities are beyond the essentials and won't be detailed here. Let's look at some other parts for your exemplary `cacti` module instead.

Enhancing the agent through plugins

The reusable classes and defines give manifests that use your module much more expressive power. Installing and configuring Cacti now works concisely, and the manifest to do this becomes very readable and maintainable.

It's time to tap into the even more powerful aspect of modules - Puppet plugins. The different types of plugins are custom facts (which were discussed in *Chapter 3, A Peek under the Hood – Facts, Types, and Providers*), parser functions, resource types, and providers. All these plugins are stored in the modules on the master and get synchronized to all the agents. The agent will not use the parser functions (they are available to the users of `puppet apply` on the agent machine once they are synchronized, however); instead the facts and resource types do most of their work on the agent. Let's concentrate on the types and providers for now - the other plugins will be discussed in dedicated sections later.

 This section can be considered optional. Many users will never touch the code for any resource type or provider – the manifests give you all the flexibility you will ever need. If you don't care for plugins, do skip ahead to the final sections about finding the Forge modules. On the other hand, if you are confident about your Ruby skills and would like to take advantage of them in your Puppet installations, read on to find the ways in which custom types and providers can help you.

While the custom resource types are functional on both the master and the agent, the provider will do all its work on the agent side. Although the resource types also perform mainly through the agent, they have one effect on the master: they enable manifests to declare resources of the type. The code not only describes what properties and parameters exist, but it can also include the validation and transformation code for the respective values. This part is invoked by the agent. Some resource types even do the synchronization and queries themselves, although there is usually at least one provider that takes care of this.

In the previous section, you implemented a defined type that did all its synchronization by wrapping some `exec` resources. By installing binaries and scripts through Puppet, you can implement almost any kind of functionality this way and extend Puppet without ever writing one plugin. This does have some disadvantages, however:

- The output is cryptic in the ideal case and overwhelming in the case of errors
- Puppet shells out to at least one external process per resource; and in many cases, multiple forks are required

In short, you pay a price, both in terms of usability and performance. Consider the `cacti::device` type. For each declared resource, Puppet will have to run an `exec` resource's `unless` query on each run (or `onlyif` when `ensure => absent` is specified). This consists of one call to a PHP script (which can be expensive) as well as several core utilities that have to parse the output. On a Cacti server with dozens or hundreds of managed devices, these calls add up and make the agent spend a lot of time forking off and waiting for these child processes.

Consider a provider, on the other hand. It can implement an `instances` hook, which will create an internal list of configured Cacti devices once during initialization. This requires only one PHP call in total, and all the processing of the output can be done in the Ruby code directly inside the agent process. These savings alone will make each run much less expensive: resources that are already synchronized will incur no penalty, because no additional external commands need to be run.

Let's take a quick look at the agent output before we go ahead and implement a simple type/provider pair. Following is the output of the `cacti::device` type when it creates a device:

```
Notice: /Stage[main]/Main/Node[agent]/Cacti::Device[Agent_VM_Debian_7]/
Exec[add-cacti-device-Agent_VM_Debian_7]/returns: executed successfully
```

The native types express such actions in a much cleaner manner, such as the output from a `file` resource:

```
Notice: /Stage[main]/Main/File[/usr/local/bin/cacti-search-graph]/ensure: created
```

Replacing a defined type with a native type

The process of creating a custom resource type with a matching provider (or several providers) is not easy. Let's go through the steps involved:

1. Naming your type.
2. Creating the resource type's interface.
3. Designing sensible parameter hooks.
4. Using resource names.
5. Adding a provider.
6. Declaring management commands.
7. Implementing the basic functionality.
8. Allowing the provider to prefetch existing resources.
9. Making the type robust during the provisioning.

Naming your type

The first important difference between the native and defined types is the naming. There is no module namespacing for the custom types like you get with the defined types, which are manifest-based. Native types from all the installed modules mingle freely, if you will. They use plain names. It would, therefore, be unwise to call the native implementation of `cacti::device` just `device` - this will easily clash with whatever notion of `devices` another module might have. The obvious choice for naming your first resource type is `cacti_device`.

The type must be completely implemented in `cacti/lib/puppet/type/cacti_device.rb`. All hooks and calls will be enclosed in a `Type.newtype` block:

```
Puppet::Type.newtype(:cacti_device) do
  @doc = <<-EOD
    Manages Cacti devices.
  EOD
end
```

The documentation string in `@doc` should be considered mandatory, and it should be a bit more substantial than this example. Consider including one or more example resource declarations. Put all the further code pieces between the `EOD` terminator and the final `end`.

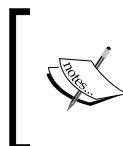
Creating the resource type interface

First of all, the type should have the `ensure` property. Puppet's resource types have a handy helper method that generates all the necessary type code for it through a simple invocation:

```
ensurable
```

With this method call in the body of the type, you add the typical `ensure` property, including all the necessary hooks. This line is all that is needed in the type code (actual implementation will follow in the provider). Most properties and parameters require more code, just like the `ip` parameter:

```
require 'ipaddr'
newparam(:ip) do
  desc "The IP address of the device."
  isrequired
  validate do |value|
    begin
      IPAddress.new(value)
    rescue ArgumentError
      fail "'#{value}' is not a valid IP address"
    end
  end
  munge do |value|
    value.downcase
  end
end
```



This should usually be an `ip` property instead, but the provider will rely on the Cacti CLI, which has no capability for changing the already configured devices. If the IP address was a property, such changes would be required in order to perform property-value synchronization.

As you can see, the IP address parameter code consists mostly of validation. Add the `require 'ipaddr'` line near the top of the file rather than inside the `Type.newtype` block.

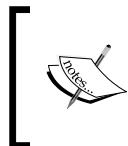
The parameter is now available for the `cacti_device` resources, and the agent will even refuse to add devices whose IP addresses are not valid. This is helpful for the users, because obvious typos in the addresses will be detected early. Let's implement the next parameter before we look at the `munge` hook more closely.

Designing sensible parameter hooks

Moving right along to the `ping_method` parameter, it accepts values only from a limited set, so validation is easy:

```
newparam(:ping_method) do
  desc "How the device's reachability is determined.
        One of `tcp` (default), `udp` or `icmp`."
  validate do |value|
    [ :tcp, :udp, :icmp ].include?(value.downcase.to_sym)
  end
  munge do |value|
    value.downcase.to_sym
  end
  defaultto :tcp
end
```

Looking at the `munge` blocks carefully, you will notice that they aim at unifying the input values. This is much less critical for the parameters than the properties, but if either of these parameters is changed to a property in a future release of your Cacti module, it will not try to sync a `ping_method` of `tcp` to `TCP`. The latter might appear if the users prefer uppercase in their manifest. Both values just become `:tcp` through munging. For the IP address, invoking `downcase` has an effect only for IPv6.



Beyond the scope of Puppet itself, the munging of a parameter's value is important as well. It allows Puppet to accept more convenient values than the subsystem being managed. For example, Cacti might not accept `TCP` as a value, but Puppet will, and it will do the right thing with it.

Using resource names

You need to take care of one final requirement: each Puppet resource type must declare a **name variable** or **namevar**, for short. This parameter will use the resource title from the manifest as its value, if the parameter itself is not specified for the resource. For example, the `exec` type has the `command` parameter for its `namevar`. You can either put the executable command into the resource title or explicitly declare the parameter:

```
exec { '/bin/true': }
# same effect:
exec { 'some custom name': command => '/bin/true' }
```

To mark one of the existing parameters as the name variable, call the `isnamevar` method in that parameter's body. If a type has a parameter called `:name`, it automatically becomes the name variable. This is a safe default.

```
newparam(:name) do
  desc "The name of the device."
  #isnamevar # → commented because automatically assumed
end
```

Adding a provider

The resource type itself is ready for action, but it lacks a provider to do the actual work of inspecting the system and performing the synchronization. Let's build it step by step, just like the type. The name of the provider need not reflect the resource type it's for. Instead, it should contain a reference to the management approach it implements. Since your provider will rely on the Cacti CLI, name it `cli`. It's fine for multiple providers to share a name if they provide functionality to different types.

Create the skeleton structure in `cacti/lib/puppet/provider/cacti_device/cli.rb`:

```
Puppet::Type.type(:cacti_device).provide(
  :cli,
  :parent => Puppet::Provider
) do
end
```

Specifying `:parent => Puppet::Provider` is not necessary, actually.

`Puppet::Provider` is the default base class for the providers. If you write a couple of similar providers for a subsystem (each catering to a different resource type), all of which rely on the same toolchain, you might want to implement a base provider that becomes the parent for all the sibling providers.

For now, let's concentrate on putting together a self-sufficient `cli` provider for the `cacti_device` type. First of all, declare the commands that you are going to need.

Declaring management commands

Providers use the `commands` method to conveniently bind executables to Ruby identifiers:

```
commands :php      => 'php'
commands :add_device => '/usr/share/cacti/cli/add_device.php'
commands :add_graphs => '/usr/share/cacti/cli/add_graphs.php'
commands :rm_device  => '/usr/share/cacti/cli/remove_device.php'
```

You won't be invoking `php` directly. It's included here because declaring commands serves two purposes:

- You can conveniently call the commands through a generated method
- The provider will mark itself as valid only if all the commands are found

So, if the `php` CLI command is not found in Puppet's search path, Puppet will consider the provider to be dysfunctional. The user can determine this error condition quite quickly through Puppet's debug output.

Implementing the basic functionality

The basic functions of the provider can now be implemented in three instance methods. The names of these methods are not magic as such, but these are the methods that the default `ensure` property expects to be available (remember that you used the `ensurable` shortcut in the type code).

The first is the method that creates a resource if it does not exist yet. It must gather all the resource parameter's values and build an appropriate call to `add_device.php`:

```
def create
  args = []
  args << "--description=#{resource[:name]}"
  args << "--ip=#{resource[:ip]}"
  args << "--ping_method=#{resource[:ping_method]}"
  add_device(*args)
end
```



Don't quote the parameter values as you would quote them on the command line. Puppet takes care of this for you. It also escapes any quotes that are in the arguments, so in this case, Cacti will receive any quotes for inclusion in the configuration. For example, this will lead to a wrong title:

```
args << "--description='#{resource[:name]}'"
```

The provider must also be able to remove or destroy an entity:

```
def destroy
  rm_device("--device-id=#{@property_hash[:id]}")
end
```

The `property_hash` variable is an instance member of the provider. Each resource gets its specific provider instance. Read on to learn how it gets initialized to include the device's ID number.

Before we get to that, let's add the final provider method in order to implement the `ensure` property. This is a query method that the agent uses to determine whether a resource is already present:

```
def exists?
  self.class.instances.find do |provider|
    provider.name == resource[:name]
  end
end
```

The `ensure` property relies on the provider class method `instances` in order to get a list of providers for all the entities on the system. It compares each of them with the `resource` attribute, which is the resource type instance for which this current provider instance is performing the work. If this is rather confusing, please refer to the diagram in the next section.

Allowing the provider to prefetch existing resources

The `instances` method is truly special - it implements the prefetching of the system resources during the provider initialization. You have to add it to the provider yourself. Some subsystems are not suitable for the mass-fetching of all the existing resources (such as the `file` type). These providers don't have an `instances` method. Enumerating the Cacti devices, on the other hand, is quite possible:

```
def self.instances
  return @instances ||= add_graphs("--list-hosts") .
    split("\n") .
    drop(1) .
    collect do |line|
      fields = line.split(/\t/, 4)
      Puppet.debug "prefetching cacti_device #{fields[3]} " +
        "with ID #{fields[0]}"
      new(:ensure => :present,
          :name    => fields[3],
          :id      => fields[0])
    end
end
```

The `ensure` value of the provider instance reflects the current state. The method creates instances for the resources that are found on the system, so for these, the value is always `present`. Also note that the result of the method is cached in the `@instances` class member variable. This is important, because the `exists?` method calls `instances`, which can happen a lot.

Puppet requires another method to perform proper prefetching. The mass-fetching you implemented through `instances` supplies the agent with a list of provider instances that represent the entities found on the system. From the master, the agent received a list of the resource type instances. However, Puppet has not yet built a relation between the resources (type instances) and providers. You need to add a `prefetch` method to the provider class in order to make this happen:

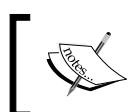
```
def self.prefetch(resources)
  instances.each do |provider|
    if res = resources[provider.name]
      res.provider = provider
    end
  end
end
```

The agent passes the `cacti_device` resources as a hash, with the resource title as the respective key. This makes lookups very simple (and quick).

This completes the `cli` provider for the `cacti_device` type. You can now replace your `cacti::device` resources with the `cacti_device` instances to enjoy improved performance and cleaner agent output:

```
node 'agent' {
  include cacti
  cacti_device { 'Puppet test agent (Debian 7)':
    ensure => present,
    ip      => $ipaddress,
  }
}
```

Note that unlike your defined type `cacti::device`, a native type will not assume a default value of `present` for its `ensure` property. Therefore, you have to specify it for any `cacti_device` resource. Otherwise, Puppet will only manage the properties of the resources that already exist and not care about whether the entity exists or not. In the particular case of `cacti_device`, this will never do anything, because there are no other properties (only parameters).



You can refer to *Chapter 6, Leveraging the Full Toolset of the Language*, on how to use resource defaults to save you from the repetition of the `ensure => present` specification.

Making the type robust during provisioning

There is yet another small issue with the `cacti` module. It is self-sufficient and handles both the installation and configuration of Cacti. However, this means that during Puppet's first run, the `cacti` package and its CLI will not be available, and the agent will correctly determine that the `cli` provider is not yet suitable. Since it is the only provider for the `cacti_device` type, any resource of this type that is synchronized before the `cacti` package will fail.

In the case of the defined type `cacti::device`, you just added the `require` metaparameters to the inner resources. To achieve the same end for the native type instances, you can work with the `autorequire` feature. Just as the files automatically depend on their containing directory, the Cacti resources should depend on the successful synchronization of the `cacti` package. Add the following block to the `cacti_device` type:

```
autorequire :package do
  catalog.resource(:package, 'cacti')
end
```

Enhancing Puppet's system knowledge through facts

When facts were introduced in *Chapter 3, A Peek under the Hood – Facts, Types, and Providers*, you got a small tour of the process of creating your own custom facts. We hinted at modules at that point, and now, we can take a closer look at how the fact code is deployed, using the example of the `cacti` module. Let's focus on native Ruby facts - they are more portable than the external facts. As the latter are easy to create, there is no need to discuss them in depth here.



For details on external facts, you can refer to the online documentation on custom facts on the Puppet Labs site at http://docs.puppetlabs.com/facter/latest/custom_facts.html#external-facts.

Facts are part of the Puppet plugins that a module can contain, just like the types and providers from the previous sections. They belong in the `lib/facter/` subtree. For the users of the `cacti` module, it might be helpful to learn which graph templates are available on a given Cacti server (once the graph management is implemented, that is). The complete list can be passed through a fact. The following code in `cacti/lib/facter/cacti_graph_templates.rb` will do just this job:

```
Facter.add(:cacti_graph_templates) do
  setcode do
    cmd = '/usr/share/cacti/cli/add_graphs.php'
```

```
Facturer::Core::Execution.exec("#{$cmd} --list-graph-templates") .
  split("\n").
  drop(1).
  collect do |line|
    line.split(/\t/) [1]
  end
end
end
```

The code will call the CLI script, skip its first line of output, and join the values from the second column of each remaining line in a list. Manifests can access this list through the global `$cacti_graph_templates` variable, just like any other fact.

Refining the interface of your module through custom functions

Functions can be of great help in keeping your manifest clean and maintainable, and some tasks cannot even be implemented without resorting to a Ruby function.

A frequent use of the custom functions (especially in Puppet 3) is input validation. You can do this in the manifest itself, but it can be a frustrating exercise because of the limitations of the language. The `stdlib` module comes with the `validate_X` functions for many basic data types, such as `validate_bool`. Typed parameters in Puppet 4 and later versions make this more convenient and natural, because for the supported variable types, no validation function is needed anymore.

As with all the plugins, the functions need not be specific to the module's domain, and they instantly become available for all the manifests. Point in case is the `cacti` module that can use the validation functions for the `cacti::device` parameters. Checking whether a string contains a valid IP address is not at all specific to Cacti. On the other hand, checking whether `ping_method` is one of those that Cacti recognizes is not that generic.

To see how it works, let's just implement a function that does the job of the `validate` and `munge` hooks from the custom `cacti_device` type for the IP address parameter of `cacti::device`. This should fail the compilation if the address is invalid; otherwise, it should return the unified address value:

```
module Puppet::Parser::Functions
  require 'ipaddr'
  newfunction(:cacti_canonical_ip, :type => :rvalue) do |args|
    ip = args[0]
```

```

begin
  IPAddr.new(ip)
rescue ArgumentError
  raise "#{@resource.ref}: invalid IP address '#{ip}'"
end
ip.downcase
end
end

```

In the exception message, `@resource.ref` is expanded to the textual reference of the offending resource type instance, such as `Cacti::Device[Edge Switch 03]`.

The following example illustrates the use of the function in the simple version of `cacti::device` without the `ensure` parameter:

```

define cacti::device($ip) {
  $cli = '/usr/share/cacti/cli'
  $c_ip = cacti_canonical_ip(${ip})
  $options = "--description='${name}' --ip='${c_ip}'"
  exec { "add-cacti-device-${name}":
    command => "${cli}/add_device.php ${options}",
    require => Class[cacti],
  }
}

```

The manifest will then fail to compile if an IP address has (conveniently) transposed digits:

```
ip => '912.168.12.13'
```

IPv6 addresses will be converted to all lowercase letters.



Puppet 4 introduced a more powerful API for defining the custom functions. Refer to *Chapter 7, New Features from Puppet 4*, to learn about its advantages.

Making your module portable across platforms

Sadly, our `cacti` module is very specific to the Debian package. It expects to find the CLI at a certain place and the Apache configuration snippet at another. These locations are most likely specific to the Debian package. It will be useful for the module to work on the Red Hat derivatives as well.

The first step is to get an overview of the differences by performing a manual installation. I chose to test this with a virtual machine running Fedora 18. The basic installation is identical to Debian, except using `yum` instead of `apt-get`, of course. Puppet will automatically do the right thing here. The `puppet::install` class also contains a CLI file, though. The Red Hat package installs the CLI in `/var/lib/cacti/cli` rather than `/usr/share/cacti/cli`.

If the module is supposed to support both platforms, the target location for the `remove_device.php` script is no longer fixed. Therefore, it's best to deploy the script from a central location in the module, while the target location on the agent system becomes a module parameter, if you will. Such values are customarily gathered in a `params` class:

```
# .../cacti/manifests/params.pp
class cacti::params {
    case $osfamily {
        'Debian': {
            $cli_path = '/usr/share/cacti/cli'
        }
        'RedHat': {
            $cli_path = '/var/lib/cacti/cli'
        }
        default: {
            fail "the cacti module does not yet support the ${osfamily} platform"
        }
    }
}
```

It is best to fail the compilation for unsupported agent platforms. The users will have to remove the declaration of the `cacti` class from their module rather than have Puppet try untested installation steps that most likely will not work (this might concern Gentoo or a BSD variant).

Classes that need to access the variable value must include the `params` class:

```
class cacti::install {
    include cacti::params
    file { 'remove_device.php':
        ensure => file,
        path   => "${cacti::params::cli_path}/remove_device.php",
        source  => 'puppet:///modules/cacti/cli/remove_device.php',
        mode     => '0755',
    }
}
```

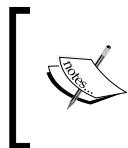
Similar transformations will be required for the `cacti::redirect` class and the `cacti::config` class. Just add more variables to the `params` class. This is not limited to the manifests, either; the facts and providers must behave in accordance with the agent platform as well.

You will often see that the `params` class is inherited rather than included:

```
class cacti($redirect = ${cacti::params::redirect})
    inherits cacti::params{
    # ...
}
```

This is done because an `include` statement in the class body won't allow the use of variable values from the `params` class as the class parameter's default values, such as the `$redirect` parameter in this example.

The portability practices are often not required for your own custom modules. In the ideal case, you won't use them on more than one platform. The practice should be considered mandatory if you intend to share them on the Forge, though. For most of your Puppet needs, you will not want to write modules anyways, but download existing solutions from the Forge instead.



In Puppet 4 and the later versions, the `params` class pattern will no longer be necessary to ship the default parameter values. There is a new data binding mechanism instead. At the time of writing this, that feature was not yet quite functional, so it is not covered in detail.

Finding helpful Forge modules

Using the web interface at <http://forge.puppetlabs.com> is very straightforward. By filling the search form with the name of the software, system, or service you need to manage, you will usually get a list of very fitting modules - often with just your search term as their name. In fact, for common terms, the number of available modules can be overwhelming.

You can get immediate feedback about the maturity and popularity of each module. A module is being actively used and maintained if:

- It has a score close to 5
- It has a version number that indicates releases past 1.0.0 (or even 0.1.0)
- Its most recent release was not too long ago, perhaps less than half a year
- It has a significant number of downloads

The latter three numbers can vary a lot, though, depending on the number of features that the module implements and how widespread its subject is. Even more importantly, just because a particular module gets much attention and regular contributions, it does not have to mean that it is the best choice for your situation.

You are encouraged to evaluate less trafficked modules as well - you can unearth some hidden gems this way. The next section details some deeper indicators of quality for you to take into consideration.

If you cannot or don't want to spend too much time digging for the best module, you can also just refer to the sidebar with the **Puppet Supported** and **Puppet Approved** modules. All modules that are featured in these categories have got a seal of quality from Puppet Labs.

Identifying modules' characteristics

When navigating to a module's details in the Forge, you are presented with its README file. An empty or very sparse documentation speaks of little care taken by the module author. A sample manifest in the README file is often a good starting point in order to put a module to work quickly.

If you are looking for a module that will enhance your agents through additional resource types and providers, look for the **Types** tab on the module details page. It can also be enlightening to click on the **Project URL** link near the top of the module description. This usually leads to GitHub. Here, you can conveniently browse not only the plugins in the `lib/` subtree, but also get a feel of how the module's manifests are structured.

Another sign of a carefully maintained module are unit tests. These are found in the `spec/` subtree. This tree does exist for most of the Forge modules. It tends to be devoid of actual tests, though. There can be test code files for all the classes and the defined types that are part of the module's manifest - these are typically in the `spec/classes/` and `spec/defines/` subdirectories, respectively. For plugins, there will ideally be unit tests in `spec/unit/` and `spec/functions/`.

Some README files of the modules contain a small greenish tag saying **Build passing**. This can turn red on occasions, stating **Build failing**. These modules use the Travis CI through GitHub, so they are likely to have at least a few unit tests.

Summary

All the development in Puppet should be done in modules, and each such module should serve as specific a purpose as possible. Most modules comprise only manifests. This suffices to provide very effective and readable node manifests that clearly and concisely express their intent by including aptly named classes and instantiating defined types.

Modules can also contain Puppet plugins in the form of resource types and providers, parser functions, or facts. All of these are usually Ruby code. External facts can be written in any language, though. Writing your own types and providers is not required, but it can boost your performance and management flexibility.

It is not necessary to write all your modules yourself. On the contrary, it's advisable to rely on the open source modules from the Puppet Forge as much as possible. The Puppet Forge is an ever-growing collection of helpful code for virtually all the systems and software that Puppet can manage. In particular, the modules that are curated by Puppet Labs are usually of very high quality. As with any open source software, you are more than welcome to add any missing requirements to the modules yourself.

After this broad view on Puppet's larger building blocks, the next chapter narrows the scope a little. Now that you have the tools to structure and compose a manifest code base, you will learn some refined techniques in order to elegantly solve some distinct problems with Puppet.

6

Leveraging the Full Toolset of the Language

After our in-depth discussions on both, the manifest structure elements (class and define) and encompassing structure (modules), you are in a great position to write manifests for all of your agents. Make sure that you get Forge modules that will do your work for you. Then, go ahead and add site-specific modules that supply composite classes for the node blocks to be used, or rather, included.

These concepts are quite a bit to take in. It's now time to decelerate a bit, lean back, and tackle simpler code structures and ideas. You are about to learn some techniques that you are not going to need every day. They can make difficult scenarios much easier, though. So, it might be a good idea to come back to this chapter again after you have spent some time in the field. You might find that some of your designs can be simplified with these tools.

Specifically, these are the techniques that will be presented:

- Templating dynamic configuration files
- Creating virtual resources
- Exporting resources to other agents
- Overriding resource parameters
- Saving redundancy using resource defaults
- Avoiding antipatterns

Templating dynamic configuration files

In the introduction, I stated that the techniques that you are now learning are not frequently required. That was true, except for this one topic. Templates are actually a cornerstone of configuration management with Puppet.

Templates are an alternative way to manage configuration files or any files, really. You have synchronized files from the master to an agent that handled some Apache configuration settings. These are not templates, technically. They are merely static files that have been prepared and are ready for carbon copying.

These static files suffice in many situations, but sometimes, you will want the master to manage very specific configuration values for each agent. These values can be quite individual. For example, an Apache server usually requires a `MaxClients` setting. Appropriate values depend on many aspects, including hardware specifications and characteristics of the web application that is being run. It would be impractical to prepare all possible choices as distinct files in the module.

Learning the template syntax

Templates make short work of such scenarios. If you are familiar with ERB templates already, you can safely skip to the next section. If you know your way around PHP or JSP, you will quickly get the hang of ERB—it's basically the same but with Ruby inside the code tags. The following template will produce `Hello, world!` three times:

```
<% ( 1 .. 3 ).each do %>
  Hello, world!
<% end %>
```

This template will also produce lots of empty lines, because the text between the `<%` and `%>` tags gets removed from the output but the final line breaks do not. To make the ERB engine do just that, change the closing tag to `-%>`:

```
<% ( 1 .. 3 ).each do -%>
  Hello, world!
<% end -%>
```

This example is not very helpful for configuration files, of course. To include dynamic values in the output, enclose Ruby expressions in a `<%=`=tag pair:

```
<% ( 1 .. 3 ).each do |index| -%>
  Hello, world #<%= index %> !
<% end -%>
```

Now, the iterator value is part of each line of the output. You can also use member variables that are prefixed with @.

These variables are populated with the values from the Puppet manifest variables:

```
<IfModule mpm_worker_module>
  ServerLimit      <%= @apache_server_limit %>
  StartServers     <%= @apache_start_servers %>
  MaxClients       <%= @apache_max_clients %>
</IfModule>
<% @apache_ports.each do |port| -%>
  Listen <%= port %>
  NameVirtualHost *:<%= port %>
<% end -%>
```

Variables that are used in a template must be defined in the same scope or scopes from which the template is used. The next section explains how this works.

In Puppet 3.x, variable values are mostly strings, arrays, or hashes. To write efficient templates, it is helpful to occasionally glance at the methods available for the respective Ruby classes. In Puppet 4, variables have more diverse values.

There are several ways to use Puppet variables in templates:

- Prefixing the variable with the @ sign: This means that the variable is global, or it was defined in the same class where the template is used. This works with Puppet 2.7, Puppet 3, and Puppet 4.
- Using the `scope.lookupvar('variablewithscopename')` function: This allows you to refer to any variable in any class of the module. Please do not look up variables in other modules; it will build an invisible dependency on the other module. The syntax works with Puppet 2, Puppet 3, and Puppet 4.
- Using `scope['variablewithscope']`: In Puppet 3, the scope hash can be used directly. The behavior is similar to `scope.lookupvar`. This will work with Puppet 3 and Puppet 4.

Using templates in practice

Templates have their own place in modules. You can place them freely in the `templates/` subtree of the module. The `template` function locates them using a simple descriptor:

```
template('cacti/apache/cacti.conf.erb')
```

This expression evaluates the content of the template found in `modules/cacti/templates/apache/cacti.conf.erb`. The first path element (without a leading slash) is the module name. The rest of the path gets translated to the `templates/` tree in the module. The function is commonly used to generate the value of a file resource's `content` property:

```
file { '/etc/apache2/conf.d/cacti.conf':
  content => template('cacti/apache/cacti.conf.erb'),
}
```

Many templates expect some variables to be defined in their scope. The easiest way to make sure that this happens is to wrap the respective `file` resource in a parameterized container. Files that are **singletons** with a well-known name, such as `/etc/ssh/sshd_config`, should be managed through a parameterized class. Configuration items that can inhabit multiple files, such as `/etc/logrotate.d/*` or `/etc/apache2/conf.d/*`, are well suited to be wrapped in defined types:

```
define logrotate::conf(
  String $pattern,
  Integer $max_days=7,
  Array $options=[]
) {
  file { "/etc/logrotate.d/$name":
    mode      => '0644',
    content   => template('logrotate/config-snippet.erb')
  }
}
```

In the preceding example, the template uses the parameters as `@pattern`, `@max_days`, and `@options`, respectively.

For a quick and dirty string transformation of your data, you can also use the `inline_template` function in your manifest. This is often found on the right-hand side of a variable assignment:

```
$comma_seperated_list = inline_template('<%= @my_array * "," %>')
```

This example assumes that the `$my_array` Puppet variable holds an array value.

Avoiding performance bottlenecks from templates

When using templates, both through the `template` and `inline_template` functions, be aware that each invocation implies a performance penalty for your Puppet master. During the compilation of the catalog, Puppet must initialize the ERB engine for any template it encounters. The ERB evaluation happens in an individual environment that is derived from the respective scope of the `template` function invocation.

It is, therefore, not even important how complex your templates are. If your manifest requires frequent expansion of a very short template, it generates an enormous overhead for each initialization. Especially in the case of an easy `inline_template` function, such as the one mentioned previously, it can be worthwhile to invest some more effort in creating a parser function instead, as seen in *Chapter 5, Extending Your Puppet Infrastructure with Modules*. A function can perform variable value transformation without incurring the cumulative penalty.

On the bright side, using templates is quite economic for the agent, who receives the whole textual file content right inside the catalog. There is no need to make an additional call to the master and retrieve file metadata. On a high-latency network, this can be a noticeable saving.

There is no silver bullet here. Don't let the performance implications deter you from turning specific configuration files into templates. Template-based solutions will often make your module more maintainable, which will usually offset performance implications—hardware is constantly getting cheaper, after all. Just don't be wasteful with frequent (and simple) expansions.

Creating virtual resources

The next technique that we are going to discuss helps you solve conflicts in your manifests and build some elegant solutions in special situations.

Remember the uniqueness constraint that was introduced in *Chapter 1, Writing Your First Manifests*; any resource must be declared at most once in a manifest. There cannot be two classes or defined type instances that declare the same `file`, `package`, or any other type of resource. Each resource must have a unique type/name combination. This applies to instances of defined types as well as native resources.

This can pose issues when multiple modules need a common resource, such as an installed package, or perhaps even independent settings in the same configuration file. A component class for such resources, as introduced in *Chapter 4, Modularizing Manifests with Classes and Defined Types*, will resolve basic conflicts of this kind. It can be included an arbitrary number of times in the same manifest.

This can be impractical when the number of shared resources is fairly large. Imagine that you find yourself in a situation where a large number of different Puppet nodes require software from a significant set of yum repositories. Puppet will happily manage the repository configuration on the agents through its `yumrepo` type. However, you don't actually want all these repositories configured on every last machine—they do incur maintenance overhead after all. It would, instead, be desirable for each node to automatically receive the configuration for all repositories it requires for its packages but not more.

When solving this using component classes, you would wrap each repository in a distinct class. The class names should closely resemble (and most likely contain) the name of the respective repositories:

```
class yumrepos::team_ninja_stable {
    yumrepo { 'team_ninja_stable':
        ensure => present,
        ...
    }
}
```

Package resources that rely on one or more such repositories will need to be accompanied by appropriate `include` statements:

```
include yumrepos::team_ninja_stable
include yumrepos::team_wizard_experimental
package { 'doombunnies':
    ensure => installed,
    require => Class[
        'yumrepos::team_ninja_stable',
        'yumrepos::team_wizard_experimental'
    ],
}
```

This is possible, but it is less than ideal. Puppet does offer an alternative way to avoid duplicate resource declarations in the form of virtual resources. It allows you to add a resource declaration to your manifest without adding the resource to the actual catalog. The virtual resource must be **realized** or **collected** for this purpose. Just like class inclusion, this realization of virtual resources can happen arbitrarily in the same manifest.

Our previous example can, therefore, use a simpler structure with just one class to declare all the yum repositories as virtual resources:

```
class yumrepos::all {
    @yumrepo { 'team_ninja_stable':
        ensure => present,
    }
    @yumrepo { 'team_wizard_experimental':
        ensure => present,
    }
}
```

The @ prefix marks the yumrepo resources as virtual. This class can be safely included by all nodes. It will not affect the catalog until the resources are realized:

```
realize(Yumrepo['team_ninja_stable'])
realize(Yumrepo['team_wizard_experimental'])
package { 'doombunnies':
    ensure => installed,
    require => [
        Yumrepo['team_ninja_stable'],
        Yumrepo['team_wizard_experimental']
    ],
}
```

The `realize` function converts the referenced virtual resources to real ones, which get added to the catalog. Granted, this is not much better than the previous code that relied on the component classes. The virtual resources do make the intent clearer, at least. Realizing them is less ambiguous than some `include` statements—a class can contain many resources and even more `include` statements.

To really improve the situation, you can introduce a defined type that can realize the repositories directly:

```
define curated::package($ensure,$repositories=[]) {
    if $repositories != [] and $ensure != 'absent' {
        realize(Yumrepo[$repositories])
    }
    package { $name: ensure => $ensure }
}
```

You can then just pass the names of `yumrepo` resources for realization:

```
curated::package { 'doombunnies':
    ensure      =>'installed',
    repositories => [
        'team_ninja_stable',
        'team_wizard_experimental',
    ],
}
```

Better yet, you can most likely prepare a hash, globally or in the scope of `curated::package`, to create the most common resolutions:

```
$default_repos = {
    'doombunnies' => [
        'team_ninja_stable',
        'team_wizard_experimental',
    ],
    ...
}
```

The `curated::package` can then look packages up if no explicit repository names are passed. Use the `has_key` function from the `puppetlabs-stdlib` module to make the lookup safer:

```
if $repositories != [] {
    realize(Yumrepo[$repositories])
}
elsif has_key($default_repos,$name) {
    $repolist = $default_repos[$name]
    realize(Yumrepo[$repolist])
}
```



This define structure is actually possible with component classes as well. The class names can be passed as a parameter or from a central data structure. The `include` function will accept variable values for class names.

Realizing resources more flexibly using collectors

Instead of invoking the `realize` function, you can also rely on a different syntactic construct, which is the `collector`:

```
Yumrepo<| title == 'team_ninja_stable' |>
```

This is more flexible than the function call at the cost of a slight performance penalty. It can be used as a reference to the realized resource(s) in certain contexts. For example, you can add ordering constraints with the chaining operator:

```
Yumrepo<| title == 'team_ninja_stable' |> -> Class['....']
```

It is even possible to change values of resource attributes during collection. There is a whole section dedicated to such overrides later in this chapter.

As the collector is based on an expression, you can conveniently realize a whole range of resources. This can be quite dynamic—sometimes, you will create virtual resources that are already being realized by a rather indiscriminate collector. Let's look at a common example:

```
User<| |>
```

With no expression, the collection encompasses all virtual resources of the given type. This allows you to collect them all, without worrying about their concrete titles or attributes. This might seem redundant, because then it makes no sense to declare the resources as virtual in the first place. However, keep in mind that the collector might appear in some select manifests only, while the virtual resources can be safely added to all your nodes.

To be a little more selective, it can be useful to group virtual resources based on their `tags`. We haven't discussed tags yet. Each resource is tagged with several identifiers. Each tag is just a simple string. You can tag a resource manually by defining the `tag` metaparameter:

```
file { '/etc/sysctl.conf': tag => 'security' }
```

The named tag is then added to the resource. Puppet implicitly tags all resources with the name of the declaring class, the containing module, and a range of other useful meta information. For example, if your user module divides the `user` resources in classes such as `administrators`, `developers`, `qa`, and other roles, you can make certain nodes or classes select all users of a given role with a collection based on the class name tag:

```
User<| tag == 'developers' |>
```

Note that the tags actually form an array. The `==` comparison will look for the presence of the `developers` element in the `tag` array in this context. Have a look at another example to make this more clear:

```
@user { 'felix':  
    ensure => present,  
    groups => [ 'power', 'sys' ],  
}  
User<| groups == 'sys' |>
```

This way, you can collect all users who are members of the `sys` group.

If you prefer function calls over the more cryptic collector syntax, you can keep using the `realize` function alongside collectors. This works without issues. Remember that each resource can be realized multiple times, even in both ways, simultaneously.

If you are wondering, the manifest for a given agent can only realize virtual resources that are declared inside this same agent's manifest. Virtual resources do not leak into other manifests. Consequently, there can be no deliberate transfer of resources from one manifest to another, either. However, there is yet another concept that allows such an exchange; this is described in the next section.

Exporting resources to other agents

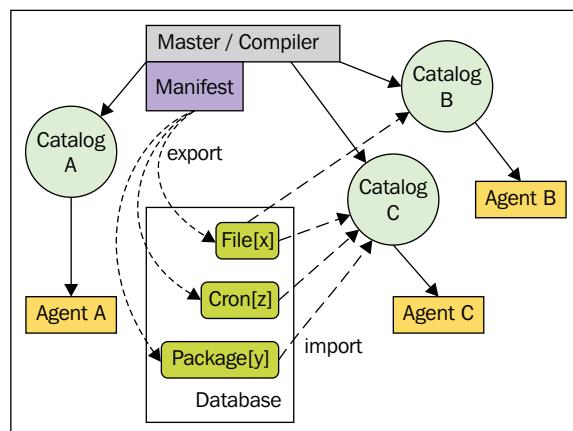
Puppet is commonly used to configure whole clusters of servers or HPC workers. Any configuration management system makes this task very efficient in comparison to manual care. Manifests can be shared between similar nodes. Configuration items that require individual customization per node are modeled individually. The whole process is very natural and direct.

On the other hand, there are certain configuration tasks that do not lend themselves well to the paradigm of the central definition of all state. For example, a cluster setup might include the sharing of a generated key or registering IP addresses of peer nodes as they become available. An automatic setup should include an exchange of such shared information. Puppet can help out with this as well.

This is a very good fit. It saves a metalayer, because you don't need to implement the setup of an information exchange system in Puppet. The sharing is secure, relying on Puppet's authentication and encryption infrastructure. There is logging and central control over the deployment of the shared configuration. Puppet retains its role as the central source for all system details; it serves as a hub for a secure exchange of information.

Exporting and importing resources

Puppet approaches the problem of sharing configuration information among multiple agent nodes by way of exported resources. The concept is simple. The manifest of node A can contain one or more resources that are purely virtual and not for realization in the manifest of this node A. Other nodes, such as B and C, can import some or all of these resources. Then, the resources become part of the catalogs of these remote nodes.



The syntax to import and export resources is very similar to that of virtual resources. An exported resource is declared by prepending the resource type name with two @ characters:

```
@@file { 'my-app-psk':
  ensure  => file,
  path    => '/etc/my-app/psk',
  content => 'nwNFGzsn9n3sDfnFANfoinaAEF',
  tag     => 'cluster02',
}
```

The importing manifests collect these resources using an expression, which is again similar to the collection of virtual resources but with double-angled brackets, < and >:

```
File<<| tag == 'cluster02' |>>
```

Tags are a very common way to take fine-grained control over the distribution of such exported resources.

Configuring the master to store exported resources

The only recommendable way to enable support for exported resources is PuppetDB. It is a domain-specific database implementation that stores different kinds of data that the Puppet master deals with during regular operation. This includes catalog requests from agents (including their valuable facts), reports from catalog applications, and exported resources.

Chapter 2, The Master and Its Agents, detailed a manual installation of the master. Let's add the PuppetDB with more style—through Puppet! On the Forge, you will find a convenient module that will make this easy:

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

On the master node, the setup now becomes a one-line invocation:

```
puppet apply -e 'include puppetdb, puppetdb::master::config'
```

As our test master uses a nonstandard SSL certificate that is named `master.example.net` (instead of its FQDN), it must be configured for `puppetdb` as well:

```
include puppetdb
class {
  'puppetdb::master::config':
    puppetdb_server => 'master.example.net'
}
```

The ensuing catalog run is quite impressive. Puppet installs the PostgreSQL backend, the Jetty server, and the actual PuppetDB package, and it configures everything and starts the services up—all in one go. After applying this short manifest, you have added a complex piece of infrastructure to your Puppet setup. You can now use exported resources for a variety of helpful tasks.

Exporting SSH host keys

For homegrown interactions between clustered machines, SSH can be an invaluable tool. File transfer and remote execution of arbitrary commands is easily possible thanks to the ubiquitous `sshd` service. For security reasons, each host generates a unique key in order to identify itself. Of course, such public key authentication systems can only really work with a trust network, or the presharing of the public keys.

Puppet can do the latter quite nicely:

```
@@sshkey { '$::fqdn':
  host_aliases => '$::hostname',
  key          => '$::sshecdsakey',
  tag          => 'san-nyc'
}
```

Interested nodes collect keys with the known pattern:

```
Sshkey<< | tag == 'san-nyc' | >>
```

Now, SSH servers can be authenticated through the respective keys that Puppet safely stores in its database. As always, the Puppet master is the fulcrum of security.



As a matter of fact, some ssh modules from the Puppet Forge will use this kind of construct to do this work for you.



Managing hosts files locally

Many sites can rely on a local DNS infrastructure. Resolving names to local IP addresses is easy with such setups. However, small networks or sites that consist of many independent clusters with little shared infrastructure will have to rely on names in `/etc/hosts` instead.

You can maintain a central hosts file per network cell, or you can make Puppet maintain each entry in each hosts file separately. The latter approach has some advantages:

- Changes are automatically distributed through the Puppet agent network
- Puppet copes with unmanaged lines in the hosts files

A manually maintained registry is prone to become outdated every once in a while. It will also obliterate local additions in any hosts files on the agent machines.

The manifest implementation of the superior approach with exported resources is very similar to the `sshkey` example from the previous section:

```
@@host { $::fqdn:  
    ip          => $::ipaddress,  
    host_aliases => [ $::hostname ],  
    tag          => 'nyc-site',  
}
```

This is the same principle, only now, each node exports its `$ipaddress` fact value alongside its name and not a public key. The import also works the same way:

```
Host<< | tag == 'nyc-site' | >>
```

Automating custom configuration items

Do you remember the Cacti module that you created during the previous chapter? It makes it very simple to configure all monitored devices in the manifest of the Cacti server. However, as this is possible, wouldn't it be even better if each node in your network was registered automatically with Cacti? It's simple: make the devices export their respective `cacti_device` resources for the server to collect:

```
@@cacti_device { $::fqdn:  
    ensure => present,  
    ip      => $::ipaddress,  
    tag     => 'nyc-site',  
}
```

The Cacti server, apart from including the `cacti` class, just needs to collect the devices now:

```
Cacti_device<< | tag == 'nyc-site' | >>
```

If one Cacti server handles all your machines, you can just omit the `tag` comparison:

```
Cacti_device<< | | >>
```

Once the module supports other Cacti resources, you can handle them in the same way. Let's look at an example from another popular monitoring solution.

Simplifying the configuration of Nagios

Puppet comes with support to manage the complete configuration of **Nagios** (and compatible versions of **Icinga**). Each configuration section can be represented by a distinct Puppet resource with types such as `nagios_host` or `nagios_service`.



There is an endeavor to remove this support from core Puppet. This does not mean that support will be discontinued, however. It will just move to yet another excellent Puppet module.

Each of your machines can export their individual `nagios_host` resources alongside their `host` and `cacti_device` resources. However, thanks to the diverse Nagios support, you can do even better.

Assuming that you have a module or class to wrap SSH handling (you are using a Forge module for the actual management, of course), you can handle monitoring from inside your own SSH server class. By adding the `export` to this class, you make sure that nodes that include the class (and only these nodes) will also get monitoring:

```
class site::ssh {
    # ...actual SSH management...
    @@nagios_service { "${::fqdn}-ssh":
        use      => 'ssh_template',
        host_name => ${::fqdn},
    }
}
```

You probably know the drill by now, but let's repeat the mantra once more:

```
Nagios_service<< | |>>
```

With this collection, the Nagios host configures itself with all services that the agent manifests create.

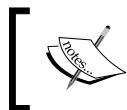


For large Nagios configurations, you might want to consider reimplementing the Nagios types yourself, using simple defines that build the configuration from templates. The native types can be slower than the `file` resources in this case, because they have to parse the whole Nagios configuration on each run. The `file` resources can be much cheaper, as they rely on content-agnostic checksums.

Maintaining your central firewall

Speaking of useful features that are not part of the core of Puppet, you can manage the rules of your `iptables` firewall, of course. You need the `puppetlabs-firewall` module to make the appropriate types available. Then, each machine can (among other useful things) export its own required port forwarding to the firewall machines:

```
@@firewall { "150 forward port 443 to ${::hostname}":
  proto      => 'tcp',
  dport      => '443',
  destination => $public_ip_address,
  jump       => 'DNAT',
  todest     => $::ipaddress,
  tag        => 'segment03',
}
```



The `$public_ip_address` value is not a Facter fact, of course. Your node will have to be configured with the appropriate information. You can refer to the next chapter for a good way to do this.



The title of a firewall rule resource conventionally begins with a three-digit index for ordering purposes. The firewall machines collect all these rules naturally:

```
Firewall<< | tag == 'segment03' | >>
```

As you can see, the possibilities for modeling distributed systems through exported Puppet resources are manifold. The simple pattern that we've iterated for several resource types suffices for a wide range of use cases. Combined with defined resource types, it allows you to flexibly enable your manifests to work together in order to form complex cluster setups with relatively little effort. The larger your clusters, the more work Puppet lifts from you through exports and collections.

Removing obsolete exports

When a node manifest stops exporting any resource, that resource's record is removed from PuppetDB once the node's catalog is compiled. This usually happens when the agent checks in with the master.

However, if you take an agent out of commission permanently, this will never happen. That's why you will need to remove those exports from the DB manually. Otherwise, other nodes will keep importing the old resources.

To clean such records from PuppetDB, use the `puppet node deactivate` command on the master server:

```
puppet node deactivate vax793.example.net
```

Overriding resource parameters

Both exported and virtual resources are declared once, and are then collected in different contexts. The syntax is very similar, as are the concepts.

Sometimes, a central definition of a resource cannot be safely realized on all of your nodes, though; for example, consider the set of all your `user` resources. You will most likely wish to manage the user ID that is assigned to each account in order to make them consistent across your networks.



This is often solved through LDAP or similar directories, but that is not possible for some sites.



Even if all accounts on almost all machines will be able to use their designated ID, there are likely to be some exceptions. On a few older machines, some IDs are probably being used for other purposes already, which cannot be changed easily. On such machines, creating users with these IDs will fail.



The accounts can be created if duplicate IDs are allowed, but that is not a solution to this problem—duplicates are usually not desirable.



Fortunately, Puppet has a convenient way to express such exceptions. To give the nonstandard UID, 2066, to the `user`, `felix`, realize the resource with an attribute value specification:

```
User[ title == 'felix' ] {
  uid => '2066'
}
```

You can pass any property, parameter, or metaparameter that applies to the resource type in question. A value that you specify this way is final and cannot be overridden again.

This language feature is more powerful than the preceding example lets on. This is because the override is not limited to virtual and exported resources. You can override any resource from anywhere in your manifest. This allows for some remarkable constructs and shortcuts.

Consider, for example, the Cacti module that you created during the previous chapter. It declares a package resource in order to make sure that the software is installed. To that end, it specifies `ensure => installed`. If any user of your module needs Puppet to keep their packages up to date, this is not adequate though. The clean solution for this case is to add some parameters to the module's classes, which allow the user to choose the `ensure` property value for the package and other resources. However, this is not really practical. Complex modules can manage hundreds of properties, and exposing them all through parameters would form a horribly confusing interface.

The override syntax can provide a simple and elegant workaround here. The manifest that achieves the desired result is very straightforward:

```
include cacti
Package<| title == 'cacti' |> { ensure => 'latest' }
```

For all its simplicity, this manifest will be hard to decipher for collaborators who are not familiar with the collector/override syntax. This is not the only problem with overrides. You cannot override the same attribute multiple times. This is actually a good thing, because any rules that resolve such conflicting overrides make it extremely difficult to predict the actual semantics of a manifest that contains multiple overrides of this kind.

Relying on this override syntax too much will make your manifests prone to conflicts. Combining the wrong classes will make the compiler stop creating the catalog. Even if you manage to avoid all conflicts, the manifests will become rather chaotic. It can be difficult to locate all active overrides for a given node. The resulting behavior of any class or define becomes hard to predict.

All things considered, it's safest to use overrides very sparingly.

[Please note that collectors are especially dangerous when used without a selector expression:



```
Package<| |> { before => Exec['send-software-list'] }
```

Not only will it realize all virtual resources of the given type. It will also force surprising attribute values on both virtual and regular resources of the same type.

Saving redundancy using resource defaults

The final language construct that this chapter introduces can save you quite some typing, or rather, it saves you from copying and pasting. Writing a long, repetitive manifest is not what costs you lots of time, of course. However, a briefer manifest is often more readable, and hence, more maintainable. You achieve this by defining resource defaults—attribute values that are used for resources that don't choose their own:

```
Mysql_grant { options => ['GRANT'],
  privileges => ['ALL'],
  tables      => '*.*',
}
mysql_grant { 'root':
  ensure => 'present',
  user   => 'root@localhost',
}
mysql_grant { 'apache':
  ensure => 'present',
  user   => 'apache@10.0.1.%',
  tables => 'application.*',
}
mysql_grant { 'wordpress':
  ensure => 'present',
  user   => 'wordpress@10.0.5.1',
  tables => 'wordpress.*',
}
mysql_grant { 'backup':
  ensure     => 'present',
  user       => 'backup@localhost',
  privileges => [ 'SELECT', 'LOCK TABLE' ],
}
```

By default, each grant should apply to all databases and comprise all privileges. This allows you to define each actual `mysql_grant` resource quite sparsely. Otherwise, you will have to specify the `privileges` property for all resources. The `options` attribute will be especially repetitive, because they are identical for all grants in this example.

Note that the `ensure` property is repetitive as well, but it was not included. It is considered good practice to exempt this attribute from resource defaults.

 The `mysql_grant` resource type is not available in core Puppet. It's part of the `puppetlabs-mysql` module on the Forge.

Despite the convenience that this approach offers, it should not be used at each apparent opportunity. It has some downsides that you should keep in mind:

- The defaults can be surprising if they apply to resources that are declared at a lexical distance from the defaults' definition (such as several screens further down the manifest file)
- The defaults transcend the inclusion of classes and instantiation of defines

These two aspects form a dangerous combination. Defaults from a composite class can affect very distant parts of a manifest:

```
class webserver {  
  File { owner => 'www-data' }  
  include apache, nginx, firewall, logging_client  
  file {  
    ...  
  }  
}
```

Files declared in the `webserver` class should belong to a default user. However, this default takes effect recursively in the included classes as well. The `owner` attribute is a property: a resource that defines no value for it just ignores its current state. A value that is specified in the manifest will be enforced by the agent. Often, you do not care about the owner of a managed file:

```
file { '/etc/motd': content => '....' }
```

However, because of the default `owner` attribute, Puppet will now mandate that this file belongs to `www-data`. To avoid this, you will have to unset the default by overwriting it with `undef`, which is Puppet's analog to the `nil` value:

```
File { owner => undef }
```

This can also be done in individual resources:

```
file { '/etc/motd': content => '...', owner => undef }
```

However, doing this constantly is hardly feasible. The latter option is especially unattractive, because it leads to more complexity in the manifest code instead of simplifying it. After all, not defining a default `owner` attribute will be the cleaner way here.



The semantics that make defaults take effect in so many manifest areas is known as **dynamic scoping**. It used to apply to variable values as well and is generally considered harmful. One of the most decisive changes in Puppet 3.0 was the removal of dynamic variable scoping, in fact. Resource defaults still use it, but it is expected that this will change in a future release as well.

Resource defaults should be used with consideration and care. For some properties such as `file mode`, `owner`, and `group`, they should usually be avoided.

Avoiding antipatterns

Speaking of things to avoid, there is a language feature that we will only address in order to advise great caution. Puppet comes with a function called `defined`, which allows you to query the compiler about resources that have been declared in the manifest:

```
if defined(File['/etc/motd']) {
    notify { 'This machine has a MotD': }
}
```

The problem with the concept is that it cannot ever be reliable. Even if the resource appears in the manifest, the compiler might encounter it later than the `if` condition. This is potentially very problematic, because some modules will try to make themselves portable through this construct:

```
if ! defined(Package['apache2']) {
    package { 'apache2':
        ensure => 'installed'
    }
}
```

The module author supposes that this resource definition will be skipped if the manifest declares `Package['apache2']` somewhere else. As explained, this method will only be effective if the block is evaluated late enough during the compiler run. The conflict can still occur if the compiler encounters the other declaration *after* this one.

The manifest's behavior becomes outright unpredictable if a manifest contains multiple occurrences of the same query:

```
class cacti {
  if !defined(Package['apache2']) {
    package { 'apache2': ensure => 'present' }
  }
}
class postfixadmin {
  if !defined(Package['apache2']) {
    package { 'apache2': ensure => 'latest' }
  }
}
```

The first block that is seen wins. This can even shift if unrelated parts of the manifest are restructured. You cannot predict whether a given manifest will use `ensure=>latest` for the `apache2` package or just use `installed`. The results become even more bizarre if such a block wants a resource removed through `ensure=>absent`, while the other does not.

The `defined` function has long been considered harmful, but there is no adequate alternative yet. The `ensure_resource` function from the `stdlib` module tries to make the scenario less problematic:

```
ensure_resource('package', 'apache2', { ensure => 'installed' })
```

By relying on this function instead of the preceding antipattern based around the `defined` function, you will avoid the unpredictable behavior of conflicting declarations. Instead, this will cause the compiler to fail when the declarations are passed to `ensure_resource`. This is still not a clean practice, though. Failed compilation is not a desirable alternative either.

Both functions should be avoided in favor of clean class structures with nonambiguous resource declarations.

Summary

A template is a frequent occurrence and is one of the best ways for Puppet to manage dynamic file content. Evaluating each template requires extra effort from the compiler, but the gain in flexibility is usually worth it. Variables in templates have to be declared using any of the three available scope lookup notations.

The concept of virtual resources is much less ubiquitous. Virtual resources allow you to flexibly add certain entities to a node's catalog. The collector syntax that is used for this can also be used to override attribute values, which works for non-virtual resources as well.

Once PuppetDB is installed and configured, you can also export resources so that other node manifests can receive their configuration information. This allows you to model distributed systems quite elegantly.

The resource defaults are just a syntactic shortcut that help keep your manifest concise. They have to be used with care, though. Some language features such as the `defined` function (and its module-based successor, which is the `ensure_resource` function) should be avoided if possible.

The next chapter gives you an overview and introduction to some of the new and enhanced features in the milestone Puppet 4 release.

7

New Features from Puppet 4

Now that we have a complete overview on the Puppet DSL and its concepts, it is time to look at the newest Puppet features that were introduced with Puppet version 4. The parser, which compiles the catalog, was basically rewritten from scratch for better performance. The milestone release also added some missing functionality and coding principles.

Puppet 4 does not only offer new functionality, but breaks with old practices and removes some functionality that is not considered a best practice anymore. This necessitates that any existing manifest code needs proper testing and potentially lots of changes to be compatible with Puppet 4.

Within this chapter, the following topics will be covered:

- Upgrading to Puppet 4
- Using the Puppet Type System
- Learning Lambdas and Functions
- Creating Puppet 4 Functions
- Leveraging the new template engine
- Handling multiline with HEREDOC
- Breaking old practices

Upgrading to Puppet 4

Let's first look at how users of the older Puppet 3 series can approach the update.



Instead of upgrading your Puppet Master machine, consider setting up a new server in parallel and migrating the service carefully. This has some advantages. For example, rolling back in case of problems is quite easy.

The new Puppet 4 version can be installed in several ways:

1. Using the Puppet Labs repositories, which will remove older Puppet packages:
This method means a hard cut without testing in advance, which is not recommended. The update to Puppet 4 should only take place after in-depth testing of your Puppet manifest code.
2. Installing as the Ruby gem extension or from tarball
This approach requires a separate Ruby 2.1 installation, which is not available on most modern Linux distributions.
3. Update to Puppet 3.8, enable and migrate to the environment path settings, and enable the future parser only on a special testing environment:
The latter solution is the smartest and most backward compatible one.



With Puppet 4 and the **All-in-One (AIO)** packaging from Puppet Labs, paths to Puppet configuration, modules, environments, and SSL certificates will change.

- Puppet 4 stores its configuration (`puppet.conf`) in `/etc/puppetlabs/puppet`.
- Hiera config will be located in `/etc/puppetlabs/hiera/hiera.yaml`.
- Puppet 4 CA and certificates can be found at `/etc/puppetlabs/puppet/ssl`.
- Puppet Code (environments and modules) are looked up in `/etc/puppetlabs/code`.

Using Puppet 3.8 and environment directories

The new parser was introduced in Puppet 3.5 alongside the old parser. To make use of the new language features, a special configuration item needed to be set explicitly. This allowed early adopters and people interested in the new technology to test the parser and check for code incompatibilities in an early stage.

On Puppet 3.x, the new parser was subject to change without further notice. Therefore it is recommended to upgrade to the latest 3.x release.

With directory environments, it is possible to specify environment specific settings within an `environment.conf` configuration file:

```
# /etc/puppet/environments/puppet_update/environment.conf
# environment config file for special puppet_update environment
parser = future
```

Next, all your puppet code needs to be put into the newly created environment path, including node classification.

On each of the different node types, it is now possible to manually run:

```
puppet agent --test --environment=puppet_update --noop
```

Check both Master and Agent output and logfiles for any errors or unwanted changes and adapt your Puppet code if necessary.

The new Puppet 4 Master



Make sure that your agents are prepared for operations with a Puppet 4 Master. See the notes about agents in the following section.



Spinning up a new Puppet Master is another approach. The following process assumes that the Puppet CA has been created using the DNS alt names setting in the `puppet.conf` file. In case no DNS alt names have been configured, it is required to completely recreate the Puppet CA.

Puppet CA needs to know about the **CN (common name)** of the Puppet Master fqdn. It is possible to provide DNS alternative names, for which the CA will also be valid.

Normally, Puppet uses the Master fqdn for the common name. But if you provide the configuration attribute `dns_alt_names` prior to generating the CA, this configuration option will be added to the CA.

It is highly recommended to configure `dns_alt_names`. Having this enabled allows you to scale up to multiple compile masters and add an additional Puppet Master for the migration process.

To find out whether DNS alt names have been added, you can use the `puppet cert` command:

```
puppet cert list -all
```

This command will print all certificates. Check for the certificate of your Puppet Master.

For example, consider the following:

```
puppet cert list --all
+ "puppetmaster.example.net" (SHA256) 7D:11:33:00:94:B3:C4:48:D2:10:B3:C7
:B0:38:71:28:C5:75:2C:61:3B:3E:63:C6:95:7C:C9:DF:59:F7:C5:BE (alt names:
"DNS:puppet", "DNS:puppet.example.net", "DNS:puppetmaster.example.net")
```

The following steps will guide you through the Puppet 4 setup. On a Debian 7 based system add the PC1 Puppet Labs repository:

```
curl -O http://apt.puppetlabs.com/puppetlabs-release-pcl-wheezy.deb
dpkg -i puppetlabs-release-pcl-wheezy.deb
apt-get update
apt-get install puppetserver puppet-agent
```

Do not start the Puppet server process yet! It is required to run the new Puppet 4 Master as CA Master, which needs the CA and certificates from the Puppet 3 Master copied over to the new Puppet 4 Master.



As of this writing, the Puppet 4 Java-based Master process is not yet available for Debian 8.



Within the next step, all Puppet agents need a change on the `puppet.conf` file. You will need to provide different settings for `ca_server` and `server`:

```
ini_setting { 'puppet_ca_server':
  path    => '/etc/puppet/puppet.conf',
  section => 'agent',
  setting => 'ca_server',
  value   => 'puppet4.example.net'
}
```



The `ini_setting` resource type is available through the `puppetlabs-inifile` module from the Forge.

Now place all your Puppet code onto the new Puppet 4 Master into an environment (`/etc/puppetlabs/code/environments/development/{manifests,modules}`).

Test your code for Puppet 4 errors by running the following command on each of your nodes:

```
puppet agent --test --noop --server puppet4.example.net --environment development
```

Change your Puppet code to fix potential errors. Once no errors and no unwanted configuration changes occur on the Puppet 4 Master and agents, your code is Puppet 4 compatible.

Updating the Puppet agent

It is important to make sure that your existing agents are prepared to operate with a master that is already at version 4. Check the following aspects:

- All agents should use the latest version of Puppet 3
- The agent configuration should specify `stringify_facts = false`

The latter step prepares you for the agent update, because Puppet 4 will always behave like that and refrain from converting all fact values to the string type.

Do make sure that you update to Puppet Server 2.1 or later. Passenger-based masters and Puppet Server 2.0 are not compatible with Puppet 3 agents.



The Puppet online documentation contains many helpful details about this update path: http://docs.puppetlabs.com/puppetserver/latest/compatibility_with_puppet_agent.html.

Testing Puppet DSL code

Another approach for verifying whether the existing Puppet code will work on Puppet 4 is unit and integration testing using `rspec-puppet` and `beaker`. This procedure is not within the scope of this book.

Whether you started fresh with Puppet 4, or used one of the preceding procedures to migrate your Puppet 3 infrastructure, it is now time to discover the benefits of the new version.

Using the type system

Older Puppet versions supported a small set of data types only: `Bool`, `String`, `Array`, and `Hash`. The Puppet DSL had almost no functionality to check for consistent variable types. Consider the following scenario.

A parameterized class enables other users of your code base to change the behavior and output of the class:

```
class ssh (
  $server = true,
) {
  if $server {
    include ssh::server
  }
}
```

This class definition checks whether the `server` parameter has been set to `true`. However, in this example, the class was not protected from wrong data usage:

```
class { 'ssh':
  server => 'false',
}
```

In this class declaration, the `server` parameter has been given a string instead of a bool value. Since the `false` string is not empty, the `if $server` condition actually passes. This is not what the user will expect.

Within Puppet 3, it was recommended to add parameter validation using several functions from the `stdlib` module:

```
class ssh (
  $server = true,
) {
  validate_bool($server)
  if $server {
    include ssh::server
  }
}
```

With one parameter only, this seems to be a good way. But what if you have many parameters? How do we deal with complex data types like hashes?

This is where the type system comes into play. The type system knows about many generic data types and follows patterns that are also used in many other modern programming languages.

Puppet differentiates between core data types and abstract data types. Core data types are the "real" data types, the ones which are mostly used in Puppet Code:

- String
- Integer, Float, and Numeric
- Boolean
- Array
- Hash
- Regexp
- Undef
- Default

In the given example, the `server` parameter should be checked to always contain a bool value. The code can be simplified to the following pattern:

```
class ssh (
  Boolean $server = true,
) {
  if $server {
    include ssh::server
  }
}
```

If the parameter is not given a Boolean value, Puppet will throw an error, explaining which parameter has a non-matching data type:

```
class { 'ssh':
  server = 'false',
}
```

The error displayed is as follows:

```
root@puppetmaster# puppet apply ssh.pp
Error: Expected parameter 'server' of 'Class[Ssh]' to have type Boolean,
got String at ssh.pp:2 on node puppetmaster.example.net
```

The `Numeric`, `Float`, and `Integer` data types have some more interesting aspects when it comes to variables and their type.

Puppet will automatically recognize `Integers`, consisting of numbers (either with or without the minus sign) and not having a decimal point.

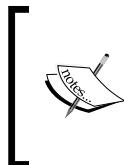
`Floats` are recognized by the decimal point. When doing arithmetic algebra on a combination of an `Integer` and a `Float`, the result will always be a `Float`.

`Floats` between -1 and 1 must be written with a leading 0 digit prior to the decimal point; otherwise, Puppet will throw an error.

Besides this, Puppet has support for the decimal, octal, and hexadecimal notation, as known from C-like languages:

- A nonzero decimal must not start with a 0
- Octal values must start with a leading 0
- Hexadecimal values have `0x` as the prefix

Puppet will automatically convert numbers into strings during the interpolation: ("Value of number: \${number}").



Puppet will not convert strings to numbers. To make this happen, you can simply add 0 to a string to convert:

```
$ssh_port = '22'  
$ssh_port_integer = 0 + $ssh_port
```



The `Default` data type is a little special. It does not directly refer to a data type, but can be used in selectors and case statements:

```
$enable_real = $enable ? {  
  Boolean => $enable,  
  String  => str2bool($enable),  
  default => fail('Unsupported value for ensure. Expected either  
  bool or string.'),  
}
```

Abstract data types are constructs that are useful for a more sophisticated or permissive Type checking:

- Scalar
- Collection

- Variant
- Data
- Pattern
- Enum
- Tuple
- Struct
- Optional
- Catalogentry
- Type
- Any
- Callable

Assume that a parameter will only accept strings from a limited set. Only checking for being of type `String` is not sufficient. In this scenario, the `Enum` type is useful, for which a list of valid values are specified:

```
class ssh (
  Boolean $server = true,
  Enum['des','3des','blowfish'] $cipher = 'des',
) {
  if $server {
    include ssh::server
  }
}
```

If the `listen` parameter is not set to one of the listed elements, Puppet will throw an error:

```
class { 'ssh':
  ciper => 'foo',
}
```

The following error is displayed:

```
puppet apply ssh.pp
Error: Expected parameter 'ciper' of 'Class[Ssh]' to have type
Enum['des','3des','blowfish'] got String at ssh.pp:2 on node
puppetmaster.example.net
```

Sometimes, it is difficult to use specific data types, because the parameter might be set to an `undef` value. Think of a `userlist` parameter that might be empty (`undef`) or set to an arbitrary array of strings.

This is what the `Optional` type is for:

```
class ssh (
  Boolean $server = true,
  Enum['des','3des','blowfish'] $cipher = 'des',
  Optional[Array[String]] $allowed_users = undef,
) {
  if $server {
    include ssh::server
  }
}
```

Again, using a wrong data type will lead to a Puppet error:

```
class { 'ssh':
  allowed_users => 'foo',
}
```

The error displayed is as follows:

```
puppet apply ssh.pp
Error: Expected parameter 'userlist' of 'Class[Ssh]' to have type
Optional[Array[String]], got String at ssh.pp:2 on node puppetmaster.
example.net
```

In the previous example, we used a data type composition. This means that data types can have more information for type checking.

Let's assume that we want to set the `ssh` service port in our class. Normally, `ssh` should run on a privileged port between 1 and 1023. In this case, we can restrict the Integer Data Type to only allow numbers between 1 and 1023 by passing additional information:

```
class ssh (
  Boolean $server = true,
  Optional[Array[String]] $allowed_users = undef,
  Integer[1,1023] $sshd_port,
) {
  if $server {
    include ssh::server
  }
}
```

As always, providing a wrong parameter will lead to an error:

```
class { 'ssh':
  sshd_port => 'ssh',
}
```

The preceding line of code gives the following error:

```
puppet apply ssh.pp
Error: Expected parameter 'sshd_port' of 'Class[Ssh]' to have type
Integer[1, 1023], got String at ssh.pp:2 on node puppetmaster.example.net
```

Complex hashes that use multiple data types are very complicated to describe using the new type system.

When using the Hash type, it is only possible to check for a hash in general, or for a hash with keys of a specific type. You can optionally verify the minimum and maximum number of elements in the hash.

The following example provides a working hash type check:

```
$hash_map = {
  'ben'      => {
    'uid'      => 2203,
    'home'     => '/home/ben',
  },
  'jones'    => {
    'uid'      => 2204,
    'home'     => 'home/jones',
  }
}
```

Notably, the home entry for user jones is missing the leading slash:

```
class users (
  Hash $users
) {
  notify { 'Valid Hash': }
}
class { 'users':
  users => $hash_map,
}
```

Running the preceding code, gives us the following output:

```
puppet apply hash.pp
Notice: Compiled catalog for puppetmaster.example.net in environment
production in 0.32 seconds
Notice: Valid hash
Notice: /Stage[main]/Users/Notify[Valid hash]/message: defined 'message'
as 'Valid hash'
Notice: Applied catalog in 0.03 seconds
```

With the preceding notation, the data type is valid. Yet there are errors inside the Hash map.

Checking content of Arrays or Hashes requires the use of another abstract data type: Tuple (used for Arrays) or Struct (used for Hashes).

However, the Struct data type will work only when the key names are from a known limited set, which is not the case in the given example.

In this special case, we have two possibilities:

- Extend the hash data type to know about the hash internal structure
- Wrap the type data into a define, which makes use of all keys using the key function (from stdlib)

The first solution is as follows:

```
class users (
  Hash[
    String,
    Struct[ { 'uid' => Integer,
              'home' => Pattern[ /^\/.*\/ ] } ]
  ] $users
) {
  notify { 'Valid hash': }
}
```

However, the error message is hard to understand when the data types are not matching:

```
puppet apply hash.pp
Error: Expected parameter 'users' of 'Class[Users]' to have type
Hash[String, Struct[{'uid'=>Integer, 'home'=>Pattern[/^\/.*\/]}]], got St
ruct[{'ben'=>Struct[{'uid'=>Integer, 'home'=>String}], 'jones'=>Struct[
{'uid'=>Integer, 'home'=>String}]] at hash.pp:32 on node puppetmaster.
example.net
```

The second solution gives a smarter hint on which data might be wrong:

```
define users::user (
  Integer      $uid,
  Pattern[/^\/.*/] $home,
) {
  notify { "User: ${title}, UID: ${uid}, HOME: ${home}": }
}
```

This defined type is then employed from within the users class:

```
class users (
  Hash[String, Hash] $users
) {
  $keys = keys($users)
  each($keys) |String $username| {
    users::user{ $username:
      uid  => $users[$username]['uid'],
      home => $users[$username]['home'],
    }
  }
}
```

With the wrong submitted data in the hash, you will receive the following error message:

```
puppet apply hash.pp
Error: Expected parameter 'home' of 'Users::User[jones]' to have type
Pattern[/^\/.*/], got String at hash.pp:23 on node puppetmaster.example.
net
```

The error message is pointing to the home parameter of the user jones, which is given in the hash.

The correct hash is as follows:

```
$hash_map = {
  'ben'    => {
    'uid'   => 2203,
    'home'  => '/home/ben',
  },
  'jones'  => {
    'uid'   => 2204,
    'home'  => '/home/jones',
  }
}
```

The preceding code produces the expected result as follows:

```
puppet apply hash.pp

Notice: Compiled catalog for puppetmaster.example.net in environment
production in 0.33 seconds

Notice: User: ben, UID: 2203, HOME: /home/ben

Notice: /Stage[main]/Users/Users::User[ben]/Notify[User: ben, UID: 2203,
HOME: /home/ben]/message: defined 'message' as 'User: ben, UID: 2203,
HOME: /home/ben'

Notice: User: jones, UID: 2204, HOME: /home/jones

Notice: /Stage[main]/Users/Users::User[jones]/Notify[User: jones, UID:
2204, HOME: /home/jones]/message: defined 'message' as 'User: jones, UID:
2204, HOME: /home/jones'

Notice: Applied catalog in 0.03 seconds
```

The preceding manifest uses the `each` function, another part of the Puppet 4 language. The next section explores it in greater detail.

Learning lambdas and functions

Functions have long been an essential part of Puppet. Due to the new type system, a complete new set of functions have been possible—functions with different behavior based on parameter data types.

To understand functions, we first have to take a look at lambdas, which are introduced in Puppet 4. Lambdas represent a snippet of Puppet code, which can be used in functions. Syntactically, lambdas consist of an optional type and at least one variable with optional defaults set, enclosed in pipe signs (`|`) followed by Puppet code inside a block of curly braces:

```
$packages = ['htop', 'less', 'vim']
each($packages) |String $package| {
    package { $package:
        ensure => latest,
    }
}
```

Lambdas are typically used on functions. The preceding example uses the `each` function on the `$packages` variable, iterating over its contents, setting the lambda variable `$package` within each iteration to the values `htop`, `less`, and `vim`, respectively. The Puppet code block afterwards uses the lambda variable inside a resource type declaration.



The Puppet code in curly braces has to ensure that no duplicate resource declaration occurs.



Since Puppet now knows about data types, you can interact and work with variables, and the data inside, in a far more elegant way.

Puppet 4 comes with a whole set of built in functions for arrays and hashes:

- `each`
- `slice`
- `filter`
- `map`
- `reduce`
- `with`

We already saw the `each` function in action. Prior to Puppet 4, one needed to wrap the desired Puppet resource types into `define` and declare the `define` type using an array:

```
class puppet_symlinks {
  $symlinks = [ 'puppet', 'facter', 'hiera' ]
  puppet_symlinks::symlinks { $symlinks: }
}

define puppet_symlinks::symlinks {
  file { "/usr/local/bin/${title}":
    ensure => link,
    target => "/opt/puppetlabs/bin/${title}",
  }
}
```

With this concept, the action (create the symlink) was put into a define type and was no longer directly visible in the manifest. The new iteration approach keeps the action in the same location:

```
class puppet_symlinks {
  $symlinks = [ 'puppet', 'facter', 'hiera' ]
  $symlinks.each | String $symlink | {
    file { "/usr/local/bin/${symlink}":
      ensure => link,
      target => "/opt/puppetlabs/bin/${symlink}",
    }
  }
}
```

Did you recognize that this time, we used another approach of using a function? In the first example, we used the Puppet 3 style function calls:

```
function($variable)
```

Puppet 4 also supports the postfix notation, where the function is appended to its parameter using a dot:

```
$variable.function
```

Puppet 4 supports both ways of using a function. This allows you to keep adhering to your code style and make use of the new functionality.

Let's run though the other functions for arrays and hashes:

- The `slice` function allows you to split up and group an array or a hash. It needs an additional parameter (integer), defining how many objects should be grouped together:

```
$array = [ '1', '2', '3', '4']
$array.slice(2) |$slice| {
    notify { "Slice: ${slice}": }
}
```

This code will produce the following output:

```
Notice: Slice: [1, 2]
Notice: Slice: [3, 4]
```

When using the slice functions on a hash, one receives the keys (according to the amount of grouped keys) and accordingly, the sub hash:

```
$hash = {
    'key 1' => {'value11' => '11', 'value12' => '12'},
    'key 2' => {'value21' => '21', 'value22' => '22'},
    'key 3' => {'value31' => '31', 'value32' => '32'},
    'key 4' => {'value41' => '41', 'value42' => '42'},
}

[hash].slice(2) |$hslice| {
    notify { "HSlice: ${hslice}": }
}
```

This will return the following output:

```
Notice: HSlice: [[key1, {value11 => 11, value12 => 12}], [key2, {value21 => 21, value22 => 22}]]  
Notice: HSlice: [[key3, {value31 => 31, value32 => 32}], [key4, {value41 => 41, value42 => 42}]]
```

- The `filter` function can be used to filter out specific entries inside an array or hash.

When used on an array, all elements are passed to the code block and the code block evaluates whether the entry does match. This is very useful if you want to filter out items of an array (for example, packages which should be installed):

```
$pkg_array = [ 'libjson', 'libjson-devel', 'libfoo', 'libfoo-devel' ]  
$dev_packages = $pkg_array.filter |$element| {  
    $element =~ /devel/  
}  
notify { "Packages to install: ${dev_packages}": }
```

This will return the following output:

```
Notice: Packages to install: [libjson-devel, libfoo-devel]
```

The behavior on hashes is different. When using hashes, one has to provide two lambda variables: key and value. You might want to only add users that have a specific gid set:

```
$hash = {  
    'jones' => {  
        'gid' => 'admin',  
    },  
    'james' => {  
        'gid' => 'devel',  
    },  
    'john' => {  
        'gid' => 'admin',  
    },  
}  
  
$user_hash = $hash.filter |$key, $value| {  
    $value['gid'] =~ /admin/  
}
```

```
$user_list = keys($user_hash)
notify { "Users to create: ${user_list}": }
```

This will return only the users from the `admin` gid:

```
Notice: Users to create: [jones, john]
```

Creating Puppet 4 functions

The Puppet 3 functions API has some limitations and is missing features. The new function API in Puppet 4 improves upon that substantially.

Some of the limitations of the old functions are as follows:

- The functions had no automatic type checking
- These functions had to have a unique name due to a flat namespace
- These functions were not private and hence could be used anywhere
- The documentation could not be retrieved without running the Ruby code

Running on Puppet 3 requires functions to be in a module in the `lib/puppet/parser/functions` directory. Therefore, people referred to these functions as **parser functions**. But this name is misleading. Functions are unrelated to the Puppet parser.

In Puppet 4, functions have to be put into a module in path `lib/puppet/functions`.

This is how you create a function that will return the hostname of the Puppet Master:

```
# modules/utils/lib/puppet/functions/resolver.rb
require 'socket'
Puppet::Functions.create_function(:resolver) do
  def resolver()
    Socket.gethostname
  end
end
```

Using `dispatch` adds type checking for attributes. Depending on desired functionality, one might have multiple `dispatch` blocks (checking for different data types). Each `dispatch` can refer to another defined Ruby method inside the function. This reference is possible by using the same names for `dispatch` and the Ruby method.

The following example code should get additional functionality; depending on the type of argument, the function should either return the hostname of the local system, or use DNS to get the hostname from an IPv4 address or the ipaddress for a given hostname:

```
require 'resolv'
require 'socket'
Puppet::Functions.create_function(:resolver) do
  dispatch :ip_param do
    param 'Pattern[/^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$/]', :ip
  end
  dispatch :fqdn_param do
    param 'Pattern[/^([a-z0-9\.\.]*$)/]', :fdqn
  end
  dispatch :no_param do
  end

  def no_param
    Socket.gethostname
  end
  def ip_param(ip)
    Resolv.getname(ip)
  end
  def fqdn_param(fqdn)
    Resolv.getaddress(fqdn)
  end
end
```

At the beginning of the file, we have to load some Ruby modules to allow the DNS name resolution and finding the local hostname.

The first two `dispatch` sections check for the data type of the parameter value and set a unique symbol. The last `dispatch` section does not check for data types, which matches when no parameter was given.

Each defined Ruby method uses the name of the according `dispatch` and executes Ruby code depending on the parameter type.

Now the resolver function can be used from inside the Puppet manifest code in three different ways:

```
class resolver {
  $localname = resolver()
```

```
notify { "Without argument resolver returns local hostname:  
  ${localname}": }  
  
$remotename = resolver('puppetlabs.com')  
notify { "With argument puppetlabs.com: ${remotename}": }  
  
$remoteip = resolver('8.8.8.8')  
notify { "With argument 8.8.8.8: ${remoteip}": }  
}
```

When declaring this class, the following output will show up:

```
puppet apply -e 'include resolver'  
Notice: Compiled catalog for puppetmaster.example.net in environment  
production in 0.35 seconds  
...  
Notice: Without argument resolver returns local hostname: puppetmaster  
  
Notice: With argument puppetlabs.com: 52.10.10.141  
  
Notice: With argument 8.8.8.8: google-public-dns-a.google.com  
  
Notice: Applied catalog in 0.04 seconds
```

With Puppet 3 functions, it was impossible to have two functions of the same name. One always had to check whether duplicate functions appeared when making use of a new module.

The Puppet 4 functions now offer the possibility of using namespacing just like classes.

Let's migrate our function into the class namespace:

```
# modules/utils/lib/puppet/functions/resolver/resolve.rb  
require 'resolv'  
require 'socket'  
Puppet::Functions.create_function(:'resolver::resolve') do  
  # the rest of the function is identical to the example given  
  # above  
end
```

In the example, the code needs to be in `resolver/lib/puppet/functions/resolver/resolve.rb` which corresponds to function name: `'resolver::resolve'`.

Functions with namespaces are invoked as usual:

```
class resolver {
    $localname = resolver::resolve()

    $remotename = resolver::resolve('puppetlabs.com')

    $remoteip = resolver::resolve('8.8.8.8')
}
```

Leveraging the new template engine

In *Chapter 6, Leveraging the Full Toolset of the Language*, we introduced templates and the ERB template engine. In Puppet 4, an alternative was added: the EPP template engine. The major differences between the template engines are as follows:

- In ERB templates, you cannot specify a variable in Puppet syntax (`$variable_name`)
- ERB templates will not accept parameters
- In EPP templates, you will use the Puppet DSL syntax instead of Ruby syntax

The EPP template engine requires scoped variables from modules:

```
# motd file - managed by Puppet
This system is running on <%= $::operatingsystem %>
```

The manifest defines the following local variable: `<%= $motd::local_variable %>`. The EPP templates also have a unique extension; they can take typed parameters. To make use of this, a template has to start with a parameter declaration block:

```
<%- | String $local_variable,
      Array  $local_array
| -%>
```

These parameters are not like variables from Puppet manifests. Instead, one must pass parameters using the `epp` function:

```
epp('template/test.epp', {'local_variable' => 'value', 'local_array'
=> ['value1', 'value2'] })
```

A template without parameters should only be used when the template is used exclusively by one module, so that it is safe to rely on the availability of Puppet variables to customize the content.

Using the EPP template function with parameters is recommended when a template is used from several places. By declaring the parameters at the beginning, it is especially clear what data the template requires.

There is a specific difference between the template engines when iterating over arrays and hashes. The ERB syntax uses Ruby code with unscoped, local variables, whereas the EPP syntax requires specifying Puppet DSL code:

```
# ERB syntax
<% @array.each do |element| -%>
<%= element %>
<% end -%>

# EPP syntax
<% $array.each |$element| { -%>
<%= $element %>
<% } -%>
```

The inline ERB function was also supplemented with inline EPP. Using the inline EPP, one can specify a small snippet of EPP code to get evaluated:

```
file {'/etc/motd':
  ensure  => file,
  content => inline_epp("Welcome to <%= $::fqdn %>\n")
}
```

Prior to Puppet 4, it was inconvenient to pass more than a small code snippet. With Puppet 4 and the HEREDOC support, complex templates in combination with `inline_epp` are easier and better readable.

Handling multiline with HEREDOC

Writing multiline file fragments in Puppet mostly resulted in code that was hard to read, mostly due to indentation. With Puppet 4, the heredoc style was added. It is now possible to specify a heredoc tag and marker:

```
$motd_content = @(EOF)
  This system is managed by Puppet
  local changes will be overwritten by next Puppet run.
EOF
```

The heredoc tag starts with an @ sign followed by arbitrary string enclosed in parenthesis. The heredoc marker is the string given in the tag.

If variables are required inside the heredoc document, the variable interpolation can be enabled by putting the tag string in double quotes. Variables inside the heredoc are written like Puppet DSL variables: a dollar sign followed by the scope and the variable name:

```
$modt_content = @("EOF")
  Welcome to ${::fqdn}.
  This system is managed by Puppet version ${::puppetversion}.
  Local changes will be overwritten by the next Puppet run
EOF
```

Normally, heredoc does not handle escape sequences. Escape sequences need to be enabled explicitly. As of Puppet 4.2, heredoc has the following escape sequences available:

- \n Newline
- \r Carriage return
- \t Tab
- \s Space
- \\$ Literal dollar sign (preventing interpolation)
- \u Unicode character
- \L Nothing (ignore line breaks in source code)

Enabled escape sequences have to be placed behind the string of the heredoc tag:

```
$modt_content = @("EOF"/\tn)
  Welcome to ${::fqdn}.\n\tThis system is managed by Puppet version
${::puppetversion}.\n\tLocal changes will be overwritten on next
Puppet run.
EOF
```

In the example, the text always starts in the first column, which makes it hard to read and stands out from the code around it, which will usually be indented by some amount of whitespace.

It is possible to strip indentation by placing whitespaces and a pipe sign in front of the heredoc marker. The pipe sign will indicate the first character of each line:

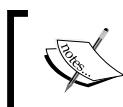
```
$motd_content = @("EOF")
  Welcome to ${::fqdn}.
  This system is managed by Puppet version ${::puppetversion}.
  Local changes will be overwritten on next Puppet run.
| EOF
```

Now heredoc and `inline_epp` can be easily combined:

```
class my_motd {
  Optional[String] $additional_content = undef
} {
  $motd_content = @("EOF")
  Welcome to <%= ${::fqdn}%>.
  This system is managed by Puppet version <%= ${::puppetversion}%>.
  Local changes will be overwritten on next Puppet run.
<% if $additional_content != undef { -%>
<%= $additional_content %>
<% } -%>
| EOF
file { '/etc/motd':
  ensure  => file,
  content => inline_epp($motd_content, { additional_content =>
    $additional_content } ),
}
}
```

Declaring this class will give the following result in the `motd` file:

```
puppet apply -e 'include my_motd'
Welcome to puppetmaster.example.net.
This system is managed by Puppet version 4.2.1.
Local changes will be overwritten on next Puppet run.
```



When using heredoc in combination with `inline_epp`, you want to take care to not quote the heredoc start tag. Otherwise, the variable substitution will take place prior to the `inline_epp` function call.

Breaking old practices

When Puppet Labs decided to work on the parser and on the new features, they also decided to remove some features that had already been deprecated for a couple of releases.

Converting node inheritance

Node inheritance has been considered good practice during older Puppet releases. To avoid too much code on the node level, a generic, nonexistent host was created (`basenode`) and the real nodes inherited from `basenode`:

```
node basenode {
  include security
  include ldap
  include base
}
node 'www01.example.net' inherits 'basenode' {
  class { 'apache': }
  include apache::mod::php
  include webapplication
}
```

This node classification is no longer supported by Puppet 4.

As of 2012, the Roles and Profiles pattern became increasingly popular, bringing new methods on how to allow smart node classification. From a technical point of view, roles and profiles are Puppet classes. The profile module wraps technical modules and adapts their usage to the existing infrastructure by providing data such as ntp servers and ssh configuration options. The role module describes system business use cases and makes use of the declared profiles:

```
class profile::base {
  include security
  include ldap
  include base
}
class profile::webserver {
  class { 'apache': }
  include apache_mod_php
}
```

```
class role::webapplication {
  include profile::base
  include profile::webserver
  include profile::webapplication
}

node 'www01.example.net' {
  include role::webapplication
}
```



The final chapter will describe the Roles and Profiles pattern in some more detail.



Dealing with bool algebra on Strings

A minor change with a huge impact is the change of empty string comparison. Prior to Puppet 4, one could test for either an unset variable or a variable containing an empty string by checking for the variable:

```
class ssh (
  $server = true,
) {
  if $server {...}
}
```

The `ssh` class behaved similar (`$server` evaluates to `true`) when used within the following different declarations:

```
include ssh
class { 'ssh': server => 'yes', }
```

Disabling the server section in the `ssh` class could be achieved by the following class declarations:

```
class { 'ssh': server => false, }
class { 'ssh': server => '', }
```

The behavior of the last example (empty string) changed in Puppet 4. The empty string now equals a true value in Boolean context just as in Ruby. If your code makes use of this way of variable checking, you need to add the check for empty string to retain the same behavior with Puppet 4:

```
class ssh (
  $server = true,
) {
  if $server and $server != '' { ... }
}
```

Using strict variable naming

Variables sometimes look like constants and exhibit the following features:

- Variables cannot be declared again
- In the scope of one node, most variables are static (hostname, fqdn, and so on)

Sometimes developers prefer to write the variables in capital letters due to the previously mentioned items, to make them look like Ruby constants.

With Puppet 4, variable names must not start with a capital letter:

```
class variables
{
  $Local_var = 'capital variable'
  notify { "Local capital var: ${Local_var}": }
}
```

Declaring this class will now produce the following error message:

```
root@puppetmaster:/etc/puppetlabs/code/environments/production/modules#
puppet apply -e 'include variables'

Error: Illegal variable name, The given name 'Local_var' does not
conform to the naming rule /^[(:)?[a-z]\w*]*[(:)?[a-z]\w*]$/
at /etc/
puppetlabs/code/environments/production/modules/variables/manifests/init.
pp:3:3 on node puppetmaster.example.net
```

Learning the new reference syntax

Due to the type system and due to the reason that Puppet 4 now takes everything as an expression, one has to name references on other declared resources properly. References now have some strict regulations:

- No whitespace between reference type and opening bracket
- The reference title (used without quotes) must not be spelled with a capital letter

The following will produce errors on Puppet 4:

```
User [Root]  
User [Root]
```

Starting with Puppet 4, references have to be written in the following pattern:

```
Type['title']
```

Our example needs to be changed to:

```
User['root']
```

Cleaning hyphens in names

Many modules (even on the module Forge) have used the hyphen in the module name. The hyphen is now no longer a string character, but a mathematical operator (subtraction). In consequence, hyphens are now strictly forbidden in the following descriptors:

- The module name
- Any class name
- Names of defined types

When using modules with hyphens, the hyphen needs to be removed or replaced with a string character (for example, the underscore).

This is possible with older versions as follows:

```
class syslog-ng { ... }  
  
include syslog-ng
```

Now, the new style is as follows:

```
class syslog_ng {  
    ...  
}  
  
include syslog_ng
```

No Ruby DSL anymore

Some people used the possibility to put .rb files as manifests inside modules. These .rb files contained Ruby code and were mostly needed for working with data. Puppet 4 now has data types that make this obsolete.

The support for these Ruby manifests has been removed in Puppet 4.

Relative class name resolution

With Puppet 3 and older, it was required to specify absolute class names in case that a local class name was identical to another module:

```
# in module "mysql"  
class mysql {  
    ...  
}  
# in module "application"  
class application::mysql {  
    include mysql  
}
```

Within the scope of the `application::` namespace, Puppet 3 would search this namespace for a `mysql` class to be included. Effectively, the `application::mysql` class would include itself. But this was not what we intended to do. We were looking for the `mysql` module instead. As a workaround, everybody was encouraged to specify the absolute path to the `mysql` module class:

```
class application::mysql {  
    include ::mysql  
}
```

This relative name resolution no longer applies in Puppet 4. The original example works now.

Dealing with different data types

Due to the reason that Puppet 3 was not aware of the different data types (mostly everything was dealt with as being of type string) it was possible to combine several different data types.

Puppet 4 now is very strict when it comes to combining different data types. The easiest example is dealing with float and integer; when adding a float and an integer, the result will be of type float.

Combining actions on different data types such as string and bool will now result in an error.

The following code will work:

```
case $::operatingsystemmajrelease {
  '8':
    include base::debian::jessie
}
}
```

On the other hand, the following code will not work:

```
if $::operatingsystemmajrelease > 7 {
  include base::debian::jessie
}
```

You will receive the following error message:

```
Error: Evaluation Error: Comparison of: String > Integer, is not
possible. Caused by 'A String is not comparable to a non String'
```

Review comparison of different Facter variables carefully. Some Facter variables such as `operatingsystemmajrelease` return data of the type string, whereas `processorcount` returns an integer value.

Summary

Upgrading to Puppet 3 should be done in a step-by-step procedure where your existing code will be evaluated using Puppet 3.8 and the new parser.

Thanks to the type system, it is now possible to deal with data in a far more elegant way directly in your Puppet DSL code. The new functions API allows you to immediately recognize to which module a function belongs by using namespaces. Similar functions can now be combined within a single file by making use of the `dispatch` method and data types, allowing a form of function overloading.

The new EPP templates offer better understanding of variable source by using the Puppet syntax for variable references. Passing parameters to templates will allow you to make use of modules in a more flexible way.

Combining EPP templates and the HEREDOC syntax will allow you to keep template code and data directly visible inside your classes.

In the upcoming final chapter, you will learn about Hiera and how it can help you bring order to a scalable Puppet code base.

8

Separating Data from Code Using Hiera

Working through the first seven chapters, you have used the basic structural elements of Puppet in numerous examples and contexts. There has been a quick demonstration of the more advanced language features, and you have a good idea of what distinguishes the manifest writing process in Puppet 4 from the earlier releases.

For all their expressive power, manifests do have some limitations. A manifest that is designed by the principles taught up to this point mixes logic with data. Logic is not only evident in control structures such as `if` and `else`, but it also just emerges from the network of classes and defines that include and instantiate one another.

However, you cannot configure a machine by just including some generic classes. Many properties of a given system are individual and must be passed as parameters. This can have maintenance implications for a manifest that must accommodate a large number of nodes. This chapter will teach you how to bring order back to such complex code bases. We will also explain how many larger sites structure the codebase as a whole. These will be our final steps in this Puppet Essentials collection:

- Understanding the need for separate data storage
- Structuring configuration data in a hierarchy
- Retrieving and using Hiera values in manifests
- Converting resources to data
- A practical example
- Debugging Hiera lookups
- Implementing the Roles and Profiles Pattern

Understanding the need for separate data storage

Looking back at what you implemented during this book so far, you managed to create some very versatile code that did very useful things in an automatic fashion. Your nodes can distribute entries for `/etc/hosts` among themselves. They register each other's public SSH key for authentication. A node can automatically register itself to a central Cacti server.

Thanks to Facter, Puppet has the information that allows effortless handling of these use cases. Many configuration items are unique to each node only because they refer to a detail (such as an IP address or a generated key) that is already defined. Sometimes, the required configuration data can be found on a remote machine only, which Puppet handles through exported resources. Such manifest designs that can rely on facts are very economical. The information has already been gathered, and a single class can most likely behave correctly for many or all of your nodes, and can manage a common task in a graceful manner.

However, some configuration tasks have to be performed individually for each node, and these can incorporate settings that are rather arbitrary and not directly derived from the node's existing properties:

- In a complex MySQL replication setup that spans multiple servers, each participant requires a unique server ID. Duplicates must be prevented under any circumstances, so randomly generating the ID numbers is not safe.
- Some of your networks might require regular maintenance jobs to be run from cron. To prevent the overlapping of the runs on any two machines, Puppet should define a starting time for each machine to ensure this.
- In server operations, you have to perform the monitoring of the disk space usage on all systems. Most disks should generate early warnings so that there is time to react. However, other disks will be expected to be almost full at most times and should have a much higher warning threshold.

When custom-built systems and software are managed through Puppet, they are also likely to require this type of micromanagement for each instance. The examples here represent only a tiny slice of the things that Puppet must manage explicitly and independently.

Consequences of defining data in the manifest

There are a number of ways in which a Puppet manifest can approach this problem of micromanagement. The most direct way is to define whole sets of classes – one for each individual node:

```
class site::mysql_server01 {
  class { 'mysql': server_id => '1', ... }
}
class site::mysql_server02 {
  class { 'mysql': server_id => '2', ... }
}
...
class site::mysql_aux01 {
  class { 'mysql': server_id => '101', ... }
}
# and so forth ...
```

This is a very high-maintenance solution for the following reasons:

- The individual classes can become quite elaborate, because all required mysql class parameters have to be used in each one
- There is much redundancy among the parameters that are, in fact, identical among all nodes
- The individually different values can be hard to spot and must be carefully kept unique throughout the whole collection of classes
- This is only really feasible by keeping these classes close together, which might conflict with other organizational principles of your code base

In short, this is the brute-force approach that introduces its own share of cost. A more economic approach would be to pass the values that are different among nodes (and only those!) to a wrapper class:

```
node 'xndp12-sql09.example.net' {
  class { 'site::mysql_server':
    mysql_server_id => '103',
  }
}
```

This wrapper can declare the `mysql` class in a generic fashion, thanks to the individual parameter value per node:

```
class site::mysql_server(
    String $mysql_server_id
) {
    class { 'mysql':
        server_id => $mysql_server_id,
        ...
    }
}
```

This is much better, because it eliminates the redundancy and its impact on maintainability. The wrinkle is that the `node` blocks can become quite messy with parameter assignments for many different subsystems. Explanatory comments contribute to the wall of text that each `node` block can become.

You can take this a step further by defining lookup tables in hash variables, outside of any `node` or `class`, on the global scope:

```
$mysql_config_table = {
    'xndp12-sql01.example.net' => {
        server_id    => '1',
        buffer_pool  => '12G',
    },
    ...
}
```

This alleviates the need to declare any variables in `node` blocks. The classes look up the values directly from the hash:

```
class site::mysql_server(
    $config = $mysql_config_table[$::certname]
) {
    class { 'mysql':
        server_id => $config['server_id'],
        ...
    }
}
```

This is pretty sophisticated and is actually close to the even better way that you will learn about in this chapter. Note that this approach still retains a leftover possibility of redundancy. Some configuration values are likely to be identical among all nodes that belong to one group, but are unique to each group (for example, preshared keys of any variety).

This requires that all servers in the hypothetical `xndp12` cluster contain some key/value pairs that are identical for all members:

```
$crypt_key_xndp12 = 'xneFG1%23ndfAWLN34a0t9w30.zges4'  
$config = {  
  'xndp12-stor01.example.net' => { $crypt_key =>  
    $crypt_key_xndp12, ... },  
  'xndp12-stor02.example.net' => { $crypt_key =>  
    $crypt_key_xndp12, ... },  
  'xndp12-sql01.example.net'  => { $crypt_key =>  
    $crypt_key_xndp12, ... },  
  ...  
}
```

This is not ideal but let's stop here. There is no point in worrying about even more elaborate ways to sort configuration data into recursive hash structures. Such solutions will quickly grow very difficult to understand and maintain anyway. The silver bullet is an external database that holds all individual and shared values. Before I go into the details of using Hiera for just this purpose, let's discuss the general ideas of hierarchical data storage.

Structuring configuration data in a hierarchy

In the previous section, we reduced the data problem to a simple need for key/value pairs that are specific to each node under Puppet management. Puppet and its manifests then serve as the engine that generates actual configuration from these minimalistic bits of information.

A simplistic approach to this problem is an `ini` style configuration file that has a section for each node that sets values for all configurable keys. Shared values will be declared in one or more general sections:

```
[mysql]
buffer_pool=15G
log_file_size=500M
...
[xndp12-sql01.example.net]
psk=xneFG1%23ndfAWLN34a0t9w30.zges4
server_id=1
```

Rails applications customarily do something similar and store their configuration in a `YAML` format. The user can define different environments, such as `production`, `staging`, and `testing`. The values that are defined per environment override the global setting values.

This is quite close to the type of hierarchical configuration that Puppet allows through its Hiera binding. The hierarchies that the mentioned Rails applications and `ini` files achieve through configuration environments are quite flat – there is a global layer and an overlay for specialized configuration. With Hiera and Puppet, a single configuration database will typically handle whole clusters of machines and entire networks of such clusters. This implies the need for a more elaborate hierarchy.

Hiera allows you to define your own hierarchical layers. There are some typical, proven examples, which are found in many configurations out there:

- The `common` layer holds default values for all agents
- A `location` layer can override some values in accordance with the data center that houses each respective node
- Each agent machine typically fills a distinct `role` in your infrastructure, such as `wordpress_appserver` or `puppetdb_server`
- Some configuration is specific to each single machine

For example, consider the configuration of a hypothetical reporting client. Your `common` layer will hold lots of presets such as default verbosity settings, the transport compression option, and other choices that should work for most machines. On the `location` layer, you ensure that each machine checks in to the respective local server – reporting should not use WAN resources.

Settings per role are perhaps the most interesting part. They allow fine-grained settings that are specific to a class of servers. Perhaps your application servers should monitor their memory consumption in very close intervals. For the database servers, you will want a closer view at hard drive operations and performance. For your Puppet servers, there might be special plugins that gather specific data.

The machine layer is very useful in order to declare any exceptions from the rule. There are always some machines that require special treatment for one reason or another. With a top hierarchy layer that holds data for each single agent, you get full control over all the data that an agent uses.

These ideas are still quite abstract, so let's finally look at the actual application of Hiera.

Configuring Hiera

The support for retrieving data values from Hiera has been built into Puppet since version 3. All you need in order to get started is a `hiera.yaml` file in the configuration directory.



Of course, the location and name of the configuration is customizable, as is almost everything that is related to configuration. Look for the `hiera_config` setting.

As the filename extension suggests, the configuration is in the YAML format and contains a hash with keys for the backends, the hierarchy, and backend-specific settings. The keys are noted as Ruby symbols with a leading colon:

```
# /etc/puppetlabs/code/hiera.yaml
:backends:
  - yaml
:hierarchy:
  - node/%{::clientcert}
  - role/%{::role}
  - location/%{::datacenter}
  - common
:yaml:
  :datadir: /etc/puppetlabs/code/environments/%{::environment}
    }/hieradata
```

Note that the value of `:backends` is actually a single element array. You can pick multiple backends. The significance will be explained later. The `:hierarchy` value contains a list of the actual layers that were described earlier. Each entry is the name of a data source. When Hiera retrieves a value, it searches each data source in turn. The `%{ }` expression allows you to access the values of Puppet variables. Use only facts or global scope variables here—anything else will make Hiera's behavior quite confusing.

Finally, you will need to include configurations for each of your backends. The configuration above uses the YAML backend only, so there is only a hash for `:yaml` with the one supported `:datadir` key. This is where Hiera will expect to find YAML files with data. For each data source, the `datadir` can contain one `.yaml` file. As the names of the sources are dynamic, you will typically create more than four or five data source files. Let's create some examples before we have a short discussion on the combination of multiple backends.

Storing Hiera data

The backend of your Hiera setup determines how you have to store your configuration values. For the YAML backend, you fill `datadir` with files that each holds a hash of values. Let's put some elements of the reporting engine configuration into the example hierarchy:

```
# /etc/puppetlabs/code/environments/production/hieradata/common.yaml
reporting::server: stats01.example.net
reporting::server_port: 9033
```

The values in `common.yaml` are defaults that are used for all agents. They are at the broad base of the hierarchy. Values that are specific to a location or role apply to smaller groups of your agents. For example, the database servers of the `postgres` role should run some special reporting plugins:

```
# /etc/puppetlabs/code/environments/production/hieradata/role/
postgres.yaml
reporting::plugins:
  - iops
  - cpupload
```

On such a higher layer, you can also override the values from the lower layers. For example, a role-specific data source such as `role/postgres.yaml` can set a value for `reporting::server_port` as well. The layers are searched from the most to the least specific, and the first value is used. This is why it is a good idea to have a node-specific data source at the top of the hierarchy. On this layer, you can override any value for each agent. In this example, the reporting node can use the loopback interface to reach itself:

```
#/etc/puppetlabs/.../hieradata/node/stats01.example.net.yaml
reporting::server: localhost
```

Each agent receives a patchwork of configuration values according to the concrete YAML files that make up its specific hierarchy.

Don't worry if all this feels a bit overwhelming. There are more examples in this chapter. Hiera also has the charming characteristic of seeming rather complicated on paper, but it feels very natural and intuitive once you try using it yourself.

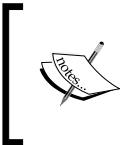
Choosing your backends

There are two built-in backends: YAML and JSON. This chapter will focus on YAML, because it's a very convenient and efficient form of data notation. The JSON backend is very similar to YAML. It looks for data in `.json` files instead of `.yaml` for each data source; these files use a different data notation format.

The use of multiple backends should never be truly necessary. In most cases, a well-thought-out hierarchy will suffice for your needs. With a second backend, data lookup will traverse your hierarchy once per backend. This means that the lowest level of your primary backend will rank higher than any layer from additional backends.

In some cases, it might be worthwhile to add another backend just to get the ability to define even more basic defaults in an alternative location—perhaps a distributed filesystem or a source control repository with different commit privileges.

Also, note that you can add custom backends to Hiera, so these might also be sensible choices for secondary or even tertiary backends. A Hiera backend is written in Ruby, like the Puppet plugins. The details of creating such a backend are beyond the scope of this book.



A particularly popular backend plugin is `eyaml`, available through the `hiera-eyaml` Ruby gem. This backend allows you to incorporate encrypted strings in your YAML data. Puppet decrypts the data upon retrieval.

You have studied the theory of storing data in Hiera at length, so it's finally time to see how to make use of this in Puppet.

Retrieving and using Hiera values in manifests

Looking up a key value in Hiera is easy. Puppet comes with a very straightforward function for this:

```
$plugins = hiera('reporting::plugins')
```

Whenever the compiler encounters such a call in the manifest of the current agent node, it triggers a search in the hierarchy. The specific data sources are determined by the hierarchy in your `hiera.yaml` file. It will almost always rely on fact values provided by the agent to make flexible data source selections.

If the named key cannot be found in the agent's hierarchy, the master aborts the catalog compilation with an error. To prevent this, it is often sensible to supply a default value with the lookup:

```
$plugins = hiera('reporting::plugins', [])
```

In this case, Puppet uses an empty array if the hierarchy mentions no plugins. On the other hand, you can purposefully omit the default value. Just like with `class` and `define` parameters, this signals that the Hiera value is required. If the user fails to supply it, Puppet will abort the manifest compilation.

Working with simple values

You have seen how to invoke the `hiera` function for value retrieval. There is really not more to it than what you have seen in the previous section, except for an optional parameter. It allows you to include an additional layer at the top of your hierarchy. If the key is found in the named data source, it will override the result from the regular hierarchy:

```
$plugins = hiera('reporting::plugins', [], 'global-overrides')
```

If the `reporting::plugins` key is found in the `global-overrides` data source, the value is taken from there. Otherwise, the normal hierarchy is searched.

Generally, assigning the retrieved value to a manifest variable is quite common. However, you can also invoke the `hiera` function in other useful contexts, such as the following:

```
@@cacti_device { '$::fqdn':
  ip => hiera('snmp_address', '$::ipaddress),
}
```

The lookup result can be handed to a resource directly as a parameter value. This is an example of how to allow Hiera to define a specific IP address per machine that should be used for a specific service. It acts as a simple way to manually override Facter's assumptions.

 It is generally safer to store Hiera lookup results in a variable first. This allows you to check their data type. In Puppet 3, you need to use an `assert` function from the `stdlib` module. Puppet 4 has an operator for this purpose:

```
$max_threads = hiera('max_threads')
if $max_threads !~ Integer {
  fail "The max_threads value must be an integer
number"
}
```

Another frequent occurrence is a parameter default that is made dynamic through a Hiera lookup:

```
define logrotate::config(
  Integer $rotations = hiera('logrotate::rotations', 7)
) {
  # regular define code here
}
```

For `logrotate::config` resources that are declared with an explicit parameter value, the Hiera value is ignored:

```
logrotate::config { '/var/log/cacti.log': rotations => 12 }
```

This can be a little confusing. Still, the pattern adds some convenience. Most agents can rely on the default. The hierarchy allows you to tune this default on multiple levels of granularity.

Binding class parameter values automatically

The concept of parameterized classes might have gotten a somewhat tarnished reputation, judging from our coverage of it so far. It allegedly makes it difficult to include classes from multiple places in the manifest, or silently allows it under shifting circumstances. While that is true, you can avoid these issues by relying on Hiera for your class parameterization needs.

Since Puppet's Version 3.2, it has been possible to choose the values for any class's parameters right in the Hiera data. Whenever you include a class that has any parameters, Puppet will query Hiera to find a value for each of them. The keys must be named after the class and parameter names, joined by a double colon. Remember the `cacti` class from *Chapter 5, Extending Your Puppet Infrastructure with Modules?* It had a `$redirect` parameter. To define its value in Hiera, add the `cacti:::redirect` key:

```
# node/cacti01.example.net.yaml
cacti:::redirect: false
```

Some classes have very elaborate interfaces – the `apache` class from the Puppet Labs Apache module accepts 49 parameters at the time of writing this. If you need many of those, you can put them into the target machine's dedicated YAML file as one coherent block of keys with values. It will be quite readable, because the `apache::` prefixes line up.

You don't save any lines compared to specifying the parameters right in the manifest, but at least the wall of options will not get in your way while you're programming in your manifests – you separated data from code.

The point that is perhaps the most redeeming for class parameterization is that each key is independent in your hierarchy. Many parameters can most likely be defined for many or all of your machines. Clusters of application servers can share some settings (if your hierarchy includes a layer on which they are grouped together), and you can override parameters for single machines as you see fit:

```
# common.yaml
apache::default_ssl_cert: /etc/puppetlabs/puppet/ssl/
certs/%{::clientcert}.pem
apache::default_ssl_key: /etc/puppetlabs/puppet/ssl/private_
keys/%{::clientcert}.pem
apache::purge_configs: false
```

The preceding example prepares your site to use the Puppet certificates for HTTPS. This is a good choice for internal services, because trust to the Puppet CA can be easily established, and the certificates are available on all agent machines. The third parameter, `purge_configs`, prevents the module from obliterating any existing Apache configuration that is not under Puppet's management.

Let's see an example of a more specific hierarchy layer which overrides this setting:

```
# role/httpsec.yaml
apache::purge_configs: true
apache::server_tokens: Minimal
apache::server_signature: off
apache::trace_enable: off
```

On machines that have the `httpsec` role, the Apache configuration should be purged so that it matches the managed configuration completely. The hierarchy of such machines also defines some additional values that are not defined in the `common` layer. The SSL settings from `common` remain untouched.

A specific machine's YAML can override keys from either layer if need be:

```
# node/sec02-sxf12.yaml
apache::default_ssl_cert: /opt/ssl/custom.pem
apache::default_ssl_key: /opt/ssl/custom.key
apache::trace_enable: extended
```

All these settings require no additional work. They take effect automatically, provided that the `apache` class from the `puppetlabs-apache` module is included.

For some users, this might be the only way in which Hiera is employed on their master, which is perfectly valid. You can even design your manifests specifically to expose all configurable items as class parameters. However, keep in mind that another advantage of Hiera is that any value can be retrieved from many different places in your manifest.

For example, if your firewalled servers are reachable through dedicated NAT ports, you will want to add this port to each machine's Hiera data. The manifest can export this value not only to the firewall server itself, but also to external servers that use it in scripts and configurations to reach the exporting machine:

```
$nat_port = hiera('site::net::nat_port')
@@firewall { "650 forward port ${nat_port} to ${::fqdn}":
  proto      => 'tcp',
  dport      => $nat_port,
  destination => hiera('site::net::nat_ip'),
```

```
    jump    => 'DNAT',
    todest  => $::ipaddress,
    tag     => hiera('site::net::firewall_segment'),
}
```

The values will most likely be defined on different hierarchical layers. `nat_port` is agent-specific and can only be defined in the `%{::fqdn}` (or `%{::clientcert}` for better security) derived data source. `nat_ip` is probably identical for all servers in the same cluster. They might share a server role. `firewall_segment` could well be identical for all servers that share the same location:

```
# stor03.example.net.yaml
site::net::nat_port: 12020
...
# role/storage.yaml
site::net::nat_ip: 198.58.119.126
...
# location/portland.yaml
site::net::firewall_segment: segment04
...
```

As previously mentioned, some of this data will be helpful in other contexts as well. Assume that you deploy a script through a defined type. The script sends messages to remote machines. The destination address and port are passed to the defined type as parameters. Each node that should be targeted can export this script resource:

```
@@site::maintenance_script {"/usr/local/bin/maint-$::fqdn"}:
  address => hiera('site::net::nat_ip'),
  port    => hiera('site::net::nat_port'),
}
```

It would be impractical to do all this in one class that takes the port and address as parameters. You would want to retrieve the same value from within different classes or even modules, each taking care of the respective exports.

Handling hashes and arrays

Some examples in this chapter defined array values in Hiera. The good news is that retrieving arrays and hashes from Hiera is not at all different from simple strings, numbers, or Boolean values. The `hiera` function will return all these values, which are ready for use in the manifest.

There are two more functions that offer special handling for such values: the `hiera_array` and `hiera_hash` functions.



The presence of these functions can be somewhat confusing. New users might assume that these are required whenever retrieving hashes or arrays from the hierarchy. When inheriting Puppet code, it can be a good idea to double-check that these derived functions are actually used correctly in a given context.

When the `hiera_array` function is invoked, it gathers all named values from the whole hierarchy and merges them into one long array that comprises all elements that were found. Take the distributed firewall configuration once more, for example. Each node should be able to export a list of rules that open ports for public access. The manifest for this would be completely driven by Hiera:

```
if hiera('site::net::nat_ip', false) {
  @@firewall { "200 NAT ports for ${::fqdn}":
    port      => hiera_array('site::net::nat_ports'),
    proto     => 'tcp',
    destination => hiera('site::net::nat_ip'),
    jump      => 'DNAT',
    todest    => $::ipaddress,
  }
}
```

Please note that the title "200 NAT ports" does not allude to the number of ports, but just adheres to the naming conventions for `firewall` resources. The numeric prefix makes it easy to maintain order. Also, note the seemingly nonsensical default value of `false` for the `site::net::nat_ip` key in the `if` clause. This forms a useful pattern, though—the resource should only be exported if `public_ip` is defined for the respective node.



Care must be taken if `false` or the empty string is a conceivable value for the key in question. In this case, the `if` clause will ignore that value. In such cases, you should use a well-defined comparison instead:

```
if hiera('feature_flag_A', undef) != undef { ... }
```

The hierarchy can then hold ports on several layers:

```
# common.yaml
nat_ports: 22
```

The SSH port should be available for all nodes that get a public address. Note that this value is not an array itself. This is fine; Hiera will include scalar values in the resulting list without any complaint.

```
# role-webserver.yaml
nat_ports: [ 80, 443 ]
```

Standalone web application servers present their HTTP and HTTPS ports to the public:

```
# tbt-backend-test.example.net.yaml
nat_ports:
  - 5973
  - 5974
  - 5975
  - 6630
```

The testing instance for your new cloud service should expose a range of ports for custom services. If it has the `webserver` role (somehow), it will lead to an export of ports 22, 80, and 443 as well as its individually chosen list.



When designing such a construct, keep in mind that the array merge is only ever-cumulative. There is no way to exclude values that were added in lower layers from the final result. In this example, you will have no opportunity to disable the SSH port 22 for any given machine. You should take good care when adding common values.

A similar alternative lookup function exists for hashes. The `hiera_hash` function also traverses the whole hierarchy and constructs a hash by merging all hashes it finds under the given Hiera key from all hierarchy layers. Hash keys in higher layers overwrite those from lower layers. All values must be hashes. Strings, arrays, or other data types are not allowed in this case:

```
# common.yaml
haproxy_settings:
  log_socket: /dev/log
  log_level: info
  user: haproxy
  group: haproxy
  daemon: true
```

These are the default settings for haproxy at the lowest hierarchy level. On web servers, the daemon should run as the general web service user:

```
# role/webserver.yaml
haproxy_settings:
  user: www-data
  group: www-data
```

When retrieved using `hiera('haproxy_settings')`, this will just evaluate to the hash, `{ 'user' => 'www-data', 'group' => 'www-data' }`. The hash at the role-specific layer completely overrides the default settings.

To get all values, create a merger using `hiera_hash('haproxy_settings')` instead. The result is likely to be more useful:

```
{ 'log_socket' => '/dev/log', 'log_level' => 'info',
  'user' => 'www-data', 'group' => 'www-data', 'daemon' => true }
```

The limitations are similar to those of `hiera_array`. Keys from any hierarchy level cannot be removed, they can only be overwritten with different values. The end result is quite similar to what you would get from replacing the hash with a group of keys:

```
# role/webserver.yaml
haproxy::user: www-data
haproxy::group: www-data
```

If you opt to do this, the data can also be easily fit to a class that can bind these values to parameters automatically. Preferring flat structures can, therefore, be beneficial. Defining hashes in Hiera is still generally worthwhile, as the next section explains.

Converting resources to data

You can now move configuration settings to Hiera and dedicate your manifest to logic. This works very seamlessly as far as classes and their parameters are concerned, because class parameters automatically retrieve their values from Hiera. For configuration that requires you to instantiate resources, you still need to write the full manifests and add manual lookup function calls.

For example, an Apache web server requires some global settings, but the interesting parts of its configuration are typically performed in virtual host configuration files. Puppet models them with defined resource types. If you want to configure an iptables firewall, you have to declare lots of resources of the `firewall` type (available through the `puppetlabs-firewall` module).

Such elaborate resources can clutter up your manifest, yet they mostly represent data. There is no inherent logic to many firewall rules (although a set of rules is derived from one or several key values sometimes). Virtual hosts often stand for themselves as well, with little or no relation to configuration details that are relevant to other parts of the setup.

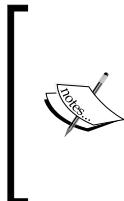
Puppet comes with yet another function that allows you to move whole sets of such resources to Hiera data. The pattern is straightforward: a group of resources of the same type are represented by a hash. The keys are resource titles, and the values are yet another layer of hashes with key/value pairs for attributes:

```
services:  
  apache2:  
    enable: true  
    ensure: running  
  syslog-nginx:  
    enable: false
```

This YAML data represents two service resources. To make Puppet add them as actual resources to the catalog, pass the hash to the `create_resources` function:

```
$resource_hash = hiera('services', {})  
create_resources('service', $resource_hash)
```

The first argument is the name of the resource type, and the second must be the hash of actual resources. There are some more aspects to this technique, but do note that with Puppet 4, it is no longer necessary to rely on the `create_resources` function.



It's useful to be aware of the basics of it anyway. It is still in broad use for existing manifests, and it is still the most compact way of converting data into resources. To learn more, refer to the online documentation at https://docs.puppetlabs.com/references/latest/function.html#create_resources



The iteration functions in the Puppet 4 language allow you to implement the preceding transformation and more:

```
$resource_hash.each |$res_title,$attributes| {  
  service { $res_title:  
    ensure => $attributes['ensure'],  
    enable => $attributes['enable'],  
  }  
}
```

This has a couple of advantages over the `create_resources` approach:

- You can perform data transformations, such as adding a prefix to string values, or deriving additional attribute values
- Each iteration can do more than just creating one resource per inner hash, for example, including required classes
- You can devise a data structure that deviates from the strict expectancies of `create_resources`
- The manifest is more clear and intuitive, especially to uninitiated readers

For creating many simple resources (such as the services in the above example), you might wish to avoid `create_resource` in Puppet 4 manifests. Just keep in mind that if you don't take advantage of doing so, you can keep the manifest more succinct by sticking to `create_resources` after all.



Puppet 4 comes with a useful tool to generate YAML data that is suitable for `create_resources`. With the following command, you can make Puppet emit service type resources that represent the set of available services on the local system, along with their current property values:

```
puppet resource -y service
```

The `-y` switch selects YAML output instead of Puppet DSL.

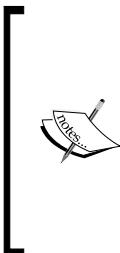
In theory, these techniques allow you to move almost all your code to Hiera data. (The next section discusses how desirable that really is.) There is one more feature that goes one step further in this direction:

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

This call gathers values from all over the hierarchy, just like `hiera_array`. The resulting array is interpreted as a list of class names. All these named classes are included. This allows for some additional consolidation in your manifest:

```
# common.yaml
classes:
  - ssh
  - syslog
...
# role-webserver.yaml
classes:
  - apache
  - logrotate
  - syslog
```

You can possibly even use `hiera_include` to declare these classes outside of any node block. The data will then affect all nodes. Additionally, from some distinct classes, you might also declare other classes via `hiera_include`, whose names are stored under a different Hiera key.



The ability to enumerate classes for each node to include is what Puppet's **External Node Classifiers (ENCs)** had originally been conceived for. Hiera can serve as a basic ENC thanks to the `hiera_include` function. This is most likely preferred over writing a custom ENC. However, it should be noted that some open source ENCs such as Foreman are quite powerful and can add much convenience; Hiera has not supplanted the concept as a whole.

The combination of these tools opens some ways for you to shrink your manifests to their essential parts and configure your machines gracefully through Hiera.

Choosing between manifest and Hiera designs

You can now move most of the concrete configuration to the data storage. Classes can be included from the manifest or through Hiera. Puppet looks up parameter values in the hierarchy, and you can flexibly distribute the configuration values there in order to achieve just the desired result for each node with minimal effort and redundancy.

This does not mean that you don't write actual manifest code anymore. The manifest is still the central pillar of your design. You will often need logic that uses the configuration data as input. For example, there might be classes that should only be included if a certain value is retrieved from Hiera:

```
if hiera('use_caching_proxy', false) {
    include nginx
}
```

If you try and rely on Hiera exclusively, you will have to add `nginx` to the `classes` array at all places in the hierarchy that set the `use_caching_proxy` flag to `true`. This is prone to mistakes. What's worse is that the flag can be overridden from `true` to `false` at a more specific layer, but the `nginx` element cannot be removed from an array that is retrieved by `hiera_include`.

It is important to keep in mind that the manifest and data should complement each other. Build manifests primarily and add lookup function calls at opportune places. Defining flags and values in Hiera should allow you (or the user of your modules) to alter the behavior of the manifest. The data should not be the driver of the catalog composition, except for places in which you replace large numbers of static resources with large data structures.

A practical example

To round things off, let's build a complete example of a module that is enhanced with Hiera. Create a `demo` module in the environment of your choice. We will go with production:

```
# /etc/puppetlabs/code/environments/production/modules/demo/manifests/
init.pp
class demo(
    Boolean $auto = false,
    Boolean $syslog = true,
    String $user = 'nobody'
) {
    file { '/usr/local/bin/demo': ... }

    if $auto {
        cron { 'auto-demo':
            user      => $user,
            command   => '/usr/local/bin/demo'
            ...
        }
        $atoms = hiera('demo::atoms', {})
        if $atoms !~ Hash[String, Hash] {
            fail "The demo::atoms value must be a hash of resource
                  descriptions"
        }
        $atoms.each |$title, $attributes| {
            demo::atom { $title:
                address => $attributes['address'],
                port     => $attributes['port'],
            }
        }
    }
}
```

This class implicitly looks up three Hiera keys for its parameters:

- `demo::auto`
- `demo::syslog`
- `demo::user`

There is also an explicit lookup of an optional `demo::atoms` hash that creates configuration items for the module. The data is converted to resources of the following defined type:

```
# /etc/puppetlabs/code/.../modules/demo/manifests/atom.pp
define demo::atom(
  String $address,
  Integer[1,65535] $port=14193
) {
  file { "/etc/demo.d/${name}":
    ensure  => 'file',
    content => "----\nhost: ${address}\nport: ${port}\n",
    mode    => '0644',
    owner   => 'root',
    group   => 'root',
  }
}
```

The module uses a default of `nobody` for `user`. Let's assume that your site does not run scripts with this account, so you set your preference in `common.yaml`. Furthermore, assume that you also don't commonly use `syslog`:

```
demo::user: automation
demo::syslog: false
```

If this user account is restricted on your guest workstations, Hiera should set an alternative value in `role/public_desktop.yaml`:

```
demo::user: maintenance
```

Now, let's assume that your cron jobs are usually managed in site modules, but not for web servers. So, you should let the `demo` module itself take care of this on web servers through the `$auto` parameter:

```
# role/webserver.yaml
demo::auto: true
```

Finally, the web server, int01-web01.example.net should be an exception. Perhaps it's supposed to run no background jobs whatsoever:

```
# int01-web01.example.net.yaml
demo::auto: false
```

Concerning configuration resources, this is how to make each machine add itself as a peer:

```
# common.yaml
demo::atoms:
  self:
    address: localhost
```

Let's assume that the Kerberos servers should not try this. Just make them override the definition with an empty hash:

```
# role/kerberos.yaml
demo::atoms: {}
```

Here is how to make the database servers also contact the custom server running on the Puppet master machine on a nonstandard port:

```
# role-dbms.yaml
demo::atoms:
  self:
    address: localhost
  master:
    address: master.example.net
    port: 60119
```

With all your Hiera data in place, you can now include the `demo` class from your `site.pp` (or an equivalent) file indiscriminately. It is often a good idea to be able to allow certain agent machines to opt out of this behavior in the future. Just add an optional Hiera flag for this:

```
# site.pp
if hiera('enable_demo', true) {
  include demo
}
```

Agents that must not include the module can be given a `false` value for the `enable_demo` key in their data now.

Debugging Hiera lookups

As you can see from the preceding example, the data that contributes to the complete configuration of any module can be rather dispersed throughout the set of your data sources. It can be challenging to determine where the respective values are retrieved from for any given agent node. It can be frustrating to trace data sources to find out why a change at some level will not take effect for some of your agents.

To help make the process more transparent, Hiera comes with a command-line tool called `hiera`. Invoking it is simple:

```
root@puppetmaster # hiera -c /etc/puppetlabs/code/hiera.yaml demo::atoms
```

It retrieves a given key using the specified configuration from `hiera.yaml`. Make sure that you use the same Hiera configuration as Puppet.

Of course, this can only work sensibly if Hiera selects the same data sources as the compiler, which uses fact values to form a concrete hierarchy. These required facts can be given right on the command line as the final parameters:

```
root@puppetmaster # hiera -c /etc/puppetlabs/code/hiera.yaml demo::atoms  
::clientcert=int01-web01.example.net ::role=webserver ::location=ny
```

This prints the `demo::atoms` value of the specified server to the console.



The fact values can also be retrieved from a YAML file or other alternative sources. Use `hiera --help` to get information about the available scenarios.



Make sure that you add the `-d` (or `--debug`) flag in order to get helpful information about the traversal of the hierarchy:

```
root@puppetmaster # hiera -d -c ...
```

Implementing the Roles and Profiles pattern

Hiera is a great tool to take control of the behavior of modules for your different agents. Most Puppet setups rely on many Forge modules to implement all the required management solutions. It is a common challenge to find efficient ways of structuring the local code.

A very successful and widespread design pattern to this end was conceived by Craig Dunn and is called the **Roles and Profiles** pattern. It defines two layers of abstraction. The outer layer is made of roles, which are defined in a way that allows for each server (or workstation) to choose exactly one role. There is no mixing—if a node has aspects of two different roles, then this merger forms a new role itself. Examples for roles can be `internal_webserver`, `key_distribution_center`, or `accounting_desktop`.

Technically, a role is just a class. It is sensible to organize your roles in a `role` module:

```
node 'falstaff.example.net' {
    include role::key_distribution_center
}
```

Do limit each `node` block to include just one role class. There should be no further `include` statements and no resource declarations. Variable declarations will be acceptable, but Hiera is almost universally preferred.

As for roles, they should comprise nothing but the inclusion of one or more profile classes. Profiles quite distinctly represent aspects of a system. In the server realm, typical profiles would be `apache_server`, `nginx_proxy`, `postgres_server`, or `ldap_master`. Just like roles, profiles should be organized in a dedicated module:

```
class role::key_distribution_center {
    include profile::heimdal_server
    include profile::firewall_internal
}
```

Profiles themselves will ideally just include a selection of modules with data from Hiera. In a `profile` class, it might also be acceptable to pass some module parameters directly in the manifest. A profile can consistently configure some of its required modules this way, without the need for possible redundancy in the Hiera data of all the nodes that use this profile.

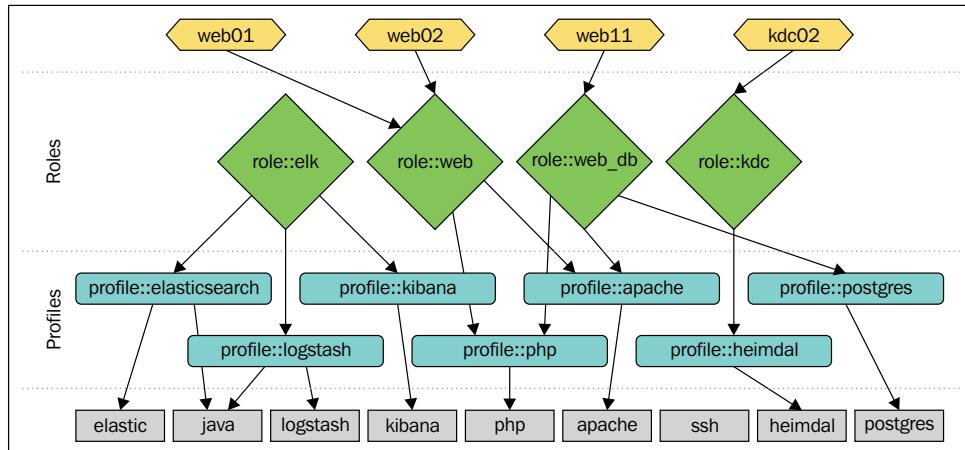
This is risky, though, because profiles can become incompatible or even impose subtle evaluation-order dependencies. Having a Hiera layer is cleaner, as it selects data sources through each node's role. At this layer, you can cleanly express configuration that should be effective for all nodes that fill this role:

```
# role/key_distribution_center.yaml
heimdal::default_realm: EXAMPLE.NET
heimdal::kdcs:
  - kdc01.example.net
  - kdc02.example.net
  - admin.example.net
```

These `heimdal` module parameters are used indirectly through the `heimdal_server` profile:

```
class profile::heimdal_server {
  include heimdal
  class { 'ssh': restricted => true }
}
```

The following diagram summarizes the pattern:



This is just a very rough sketch of the principles behind the Roles and Profiles pattern. Craig has put up a comprehensive description on his blog at <http://www.craigdunn.org/2012/05/239/>, and the design has since been adopted by many users.

When setting up your first larger Puppet installation, it is a good idea to adhere to the pattern from day one, because it will allow you to scale your manifests without getting tangled up in complicated structures.

Summary

Hiera is a tool that stores and retrieves data in a hierarchical fashion. Each retrieval uses a distinct data source from each hierarchy layer and traverses your hierarchy from the most to the least specific. The hierarchy is defined by the user, as an array in a YAML file.

Puppet has Hiera support built in, and you can use it to separate data from code. From manifests, you will mainly perform lookups through the `hiera` function. The respective entries will rely on fact values in most cases.

Another common way to employ Hiera through Puppet is to name the Hiera keys in the `<class-name>::<parameter-name>` format. When including a parameterized class, Puppet will look for such keys in Hiera. If the manifest does not supply a parameter value, Puppet automatically binds the value from Hiera to the respective parameter.

Manifests that boast large numbers of static resources can be cleaned up by converting the declarations to hashes and using the `create_resources` or `each` function to declare resources from the data.

When building and maintaining Puppet code bases, it is a good idea to implement the Roles and Profiles pattern. It makes you define roles to cover all of your machine use cases. The roles mix and match profile classes, which are basically collectors of classes from custom and open source modules that manage actual resources.

This concludes our tour of *Puppet Essentials*. We have covered quite some ground, but as you can imagine, we barely scratched the surface of some of the topics, such as provider development or exploiting PuppetDB. What you have learned will most likely satisfy your immediate requirements. For information beyond these lessons, don't hesitate to look up the excellent online documentation at <https://docs.puppetlabs.com/>, or join the community and ask your questions in chat or on the mailing list.

Thanks for reading, and have lots of fun with Puppet and its family of management tools.

Module 2

Extending Puppet, Second Edition

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

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.
- **Factor**: 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.
- **Gepetto**: 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.

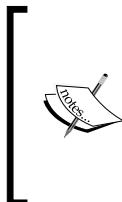
To look for the more appropriate packages for our infrastructure, we should use Puppet Labs repositories.

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, derived 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: Retrieving 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 $(facter 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`, `hpx`, `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 $(facter 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:

- **Facter** (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:

```
facter -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 facter 1.x, only plain values were available; facter 2.x introduces structured values, so any fact can contain arrays or hashes. Facter is replaced by cFacter 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:

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

External facts

External facts, supported since Puppet 3.4/Facter 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 FDQN 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.

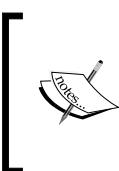
Meta parameters

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 of 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 resources 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:

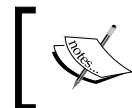
```
mysql::conf { ... }
```

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: Hiera 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.

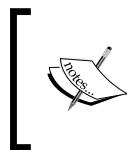
In this chapter, we will cover the following topics:

- 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
:gpg:
  :datadir: /etc/puppetlabs/code/environments/%{environment}/hieradata
  :key_dir: /etc/puppetlabs/gpgkeys

```

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
  - gpg

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

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

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

:gpg:
  :key_dir: /etc/puppetlabs/code/gpgkeys
  :datadir: /etc/puppetlabs/code/gpgdata

```

 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 `hiera.yaml` file, the files that contain Hiera data must be placed in these directories.

Let's see Hiera 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 Hiera-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 Hiera lookup for the `ldap_servers` key will still return the common value, as follows:

```
hiera 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:

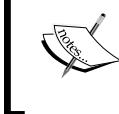
```
hiera ldap_servers env=test
["192.168.0.31"]
```

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

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



We have seen that, by default, Hiera 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.

Hiera 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):

```
hiera -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

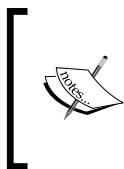
{ "al"=>{ "home"=>"/home/al", "comment"=>"Al" },
  "jenkins"=>{ "ensure"=>"absent" },
  "qa"=>{ "home"=>"/home/qa", "comment"=>"QA Tester" },
  "openldap"=>{ "groups"=>"apps" } }
```

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 Hiera data, we can manage all the users of our nodes.



We can tune how Hiera 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: Hiera 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 Hiera 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 => hiera('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 `hiera-file` we can use this syntax:

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

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

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

First of all, create a GPG key, with the following command:

```
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 `hiera-gpg` doesn't support them).

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 `hiera` to get the key's value from the encrypted files:

```
hiera mysql::root_password -d  
  
DEBUG: <datetime>: Hiera YAML backend starting  
DEBUG: <datetime>: Looking up mysql::root_password in YAML backend  
DEBUG: <datetime>: Looking for data source common  
DEBUG: <datetime>: [gpg_backend]: Loaded gpg_backend  
DEBUG: <datetime>: [gpg_backend]: Lookup called, key mysql::root_  
password resolution type is priority  
DEBUG: <datetime>: [gpg_backend]: GNUPGHOME is /root/.gnupg  
DEBUG: <datetime>: [gpg_backend]: loaded cipher: /etc/puppet/gpgdata/  
common.gpg
```

```

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

mysql::root_password: 'V3ry$3cr3T!'

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 `hiera-eyaml` works, as it is more used and maintained than `hiera-gpg`.

We install gem:

```
gem install hiera-eyaml
```

We edit the `hiera.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 `hiera` 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,MIIBeQYJKoZIhvCNQcDoIIBajCCAWYCAQAxggEhMII [...]  
+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 Hiera-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 extensibility, 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
```

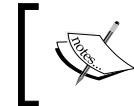
Also, we have to edit the `pg_hba.conf` server to allow access from our PuppetDB host (here, it is `10.42.42.30`, but it could be `127.0.0.1` if PostgreSQL and PuppetDB are on the same host):

```
# 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-ttl` 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
```

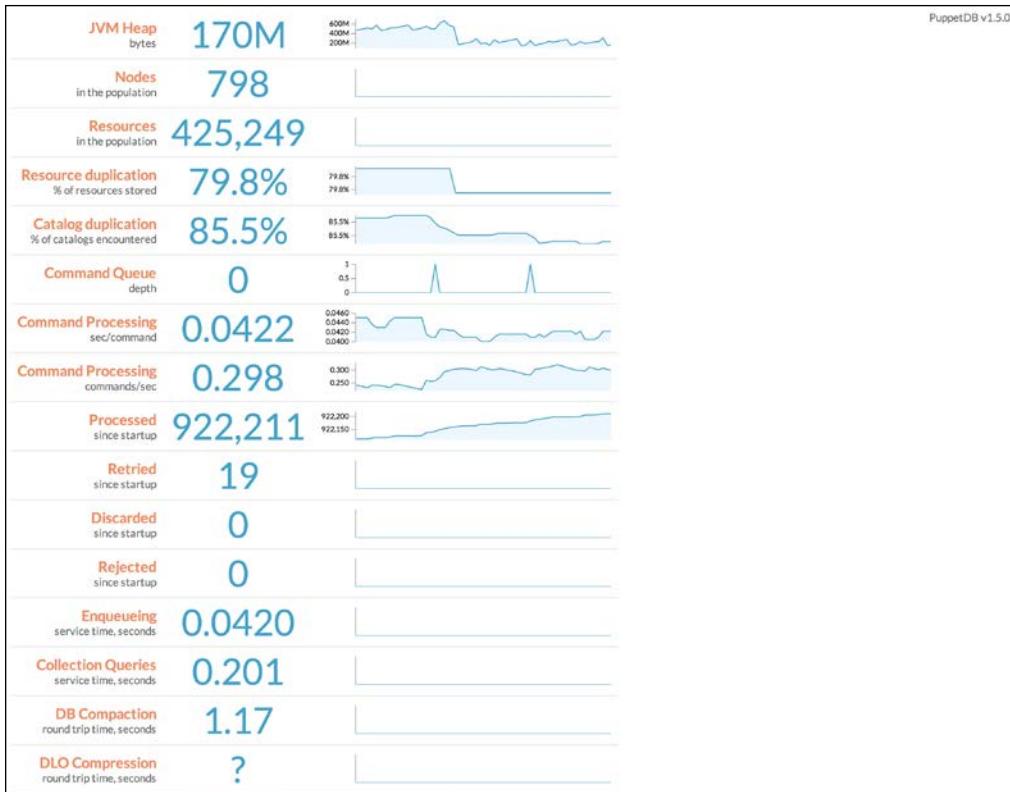
Dashboards

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 firewalls 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}d13elf5c099082afbe8a5ed9d4695beb",  
  
  "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-domain>: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 Hiera 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 `~` (regexp 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 ENCs, 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 Hiera 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.

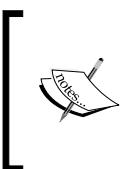
The Foreman really deserves at least one book by itself, so here we will just summarize its features and explore how it can fit in to a Puppet architecture.

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.

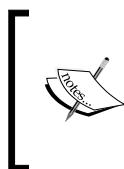
In profiles, we can manage resource and class ordering, we can initialize variables and use them as values for arguments in the declared classes, and we can generally benefit from having an extra layer of abstraction:

```
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 Hiera-only setup. This is a somewhat extreme approach. Everything is managed by Hiera: 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 Hiera.

The answer is probably yes; even in a similar, strongly Hiera-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 Hiera without adding a specific entry in the hierarchy.

One important point to consider when we move most of our logic to Hiera 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 Hiera 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 Hiera, 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 Hiera 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 Hiera 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 Hiera.

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, Hiera, 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' }
}

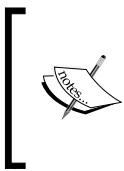
# Env is based on hostname or (sub) domain
if 'devel' in $::fqdn { $env = 'devel' }
elsif 'test' in $::fqdn { $env = 'test' }
elsif 'qa' in $::fqdn { $env = 'qa' }
else { $env = 'prod' }

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 ? {
        /(?i:Debian|Ubuntu|Mint)/      => 'apache2',
        /(?i: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',
) {
```

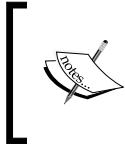
These options expose both the `install` method to use, the name of the installation class (so that it can be overridden), the URL from which to retrieve the archive and the destination to install it in.

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 to 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, ENCs 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 <https://github.com/openstack/puppet-<component>>, so, for example, Nova's module can be found at <https://github.com/openstack/puppet-nova>.

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 `hiera()` 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 time, 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.
yml.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:

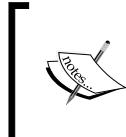
```
$real_elasticsearch_server = $elasticsearch_server ? {
    ''      => query_nodes('Class[elasticsearch]',ipaddress),
    default => $elasticsearch_server,
}
```

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.lookupvar('stack::logstash::syslog_server') %>:<%= scope.lookupvar('stack::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 Hiera data 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 reliant 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 main 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.

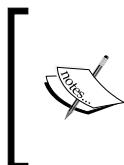
If such setups have to be done only once on a system (during installation time), and are supposed to be done on very few systems that have a medium to long life, we may question if it makes sense to try to do it with Puppet, spending several hours on what can be quickly done manually in few minutes.

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 about 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 **Gepetto** 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 **Gepetto**, 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.

Gepetto

Gepetto (<http://puppetlabs.github.io/gepetto/>) 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 Gepetto 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 Gepetto **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.github.com/tpope/vim-pathogen/master/autoload/pathogen.vim>.

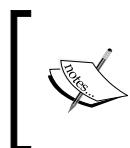
Then, be sure to have Pathogen installed in the `~/.vimrc` of our home directory:

```
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/rodjek/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.

RBCCommons (<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
```

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 Hiera 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 puppet 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 **Gepetto**, 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 <puppetmaster 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
SSLProtocolall -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/default/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_resctype_title on resources (exported, resctype,
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 ENV_PUPPET_BASEDIR=/usr/src
```

Now, we can test Puppet prepending `envpuppet` to any Puppet, Facter, or Hiera 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 Hiera 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 fileserver 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`, `hiera`, `regsubst`, `versioncmp`, and `inline_template`.
- `:statement` functions perform an action without returning any value. Samples from Puppet core are `include`, `fail`, `hiera_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
```

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 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 Abstracton 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 defines 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(:pgsql_user) do
  desc "The system user account under which the psql command should
be executed."
  defaultto("postgres")
  end
[...]
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+$/i
    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   => $pgsql_user,
  psql_group  => $group,
  psql_path   => $pgsql_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
  [...]
```

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"
```

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_psycopg` 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_psycopg).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
```

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 or 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
```

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 indirectors 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/forge'
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:

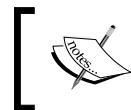
```
node 'switch01.example42.lan' {
    # Definition of the Device, needed for each device
    cisco_device { 'switch01.example42.lan':
        ensure => present,
    }

    # 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 fact, 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 hcen 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.

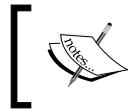
Cloud provisioning on AWS

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 reimplementation, 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 reimplementation 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 `msp` 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 Facter 3, much faster, lightweight, and with more features. A full reimplementation 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.

Module 3

Mastering Puppet, Second Edition

Master Puppet for configuration management of your systems in an enterprise deployment

1

Dealing with Load/Scale

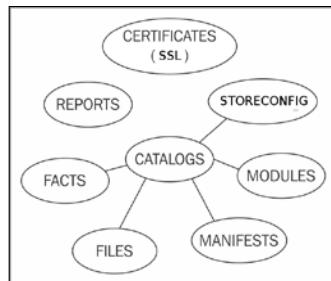
A large deployment will have a large number of nodes. If you are growing your installation from scratch, you might have to start with a single Puppet master. At a certain point in your deployment, a single Puppet master just won't cut it—the load will become too great. In my experience, this limit is around 600 nodes. Puppet agent runs begin to fail on the nodes and catalogs fail to compile. There are two ways to deal with this problem: divide and conquer or conquer by dividing.

That is, we can either split up our Puppet master, dividing the workload among several machines, or we can make each of our nodes apply our code directly using Puppet agent (this is known as a **masterless** configuration). We'll examine each of these solutions separately.

Divide and conquer

When you start to think about dividing up your `puppetserver`, the main thing to realize is that many parts of Puppet are simply HTTP TLS transactions. If you treat these things as a web service, you can scale up to any size required, using HTTP load balancing techniques.

Puppet is a web service. There are several different components supporting that web service, as shown in the following diagram:



Each of the different components in your Puppet infrastructure (SSL CA, reporting, storeconfigs, and catalog) compilation can be split up into their own server or servers, as explained in the following sections.

Certificate signing

Unless you are having issues with certificate signing consuming too many resources, it's simpler to keep the signing machine as a single instance, possibly with a hot spare. Having multiple certificate signing machines means that you have to keep certificate revocation lists synchronized.

Reporting

Reporting should be done on a single instance if possible. Reporting options will be covered in *Chapter 7, Reporting and Orchestration*.

Storeconfigs

Storeconfigs should be run on a single server; storeconfigs allows for exported resources and is optional. The recommended configuration for storeconfigs is PuppetDB, which can handle several thousand nodes in a single installation.

Catalog compilation

Catalog compilation is one task that can really bog down your Puppet installation. Splitting compilation among a pool of workers is the biggest win to scale your deployment. The idea here is to have a primary point of contact for all your nodes—the load balancer. Then, using proxying techniques, the load balancer will direct requests to specific worker machines within your Puppet infrastructure. From the perspective of the nodes checking into Puppet master, all the interaction appears to come from the main load balancing machine.

When nodes contact the Puppet master, they do so using an HTTP REST API, which is TLS encrypted. The resource being requested by a node may be any of the accepted REST API calls, such as catalog, certificate, resource, report, file_metadata, or file_content. A complete list of the HTTP APIs is available at http://docs.puppetlabs.com/guides/rest_api.html.

When nodes connect to the Puppet master, they connect to the master service. In prior versions of Puppet (versions 3.6 and older), the accepted method to run the Puppet master service was through the Passenger framework. In Puppet 3.7 and above, this was replaced with a new server, `puppetserver`. Puppet version 4 and above have deprecated Passenger; support for Passenger may be completely removed in a future release. `puppetserver` runs Puppet as a JRuby process within a JVM that is wrapped by a Jetty web server. There are many moving parts in the new `puppetserver` service, but the important thing is that Puppet Labs built this service to achieve better performance than the older Passenger implementation. A Puppet master running the `puppetserver` service can typically handle around 5,000 individual nodes; this is a vast improvement.

A quick word on versions, Puppet has now changed how they distribute Puppet. Puppet is now distributed as an all-in-one package. This package includes the required Ruby dependencies all bundled together. This new packaging has resulted in a new package naming scheme, named Puppet collections or PC. Numbering begins at 1 for the PC packages, so you will see PC1 as the package and repository name, the version of Puppet contained within those packages is version 4. Additionally, Puppet Enterprise has changed its name to a year based system; the first release of that series was 2015.1, which had a PC release of 1.2.7. More information on Puppet collections can be found at <https://puppetlabs.com/blog/welcome-puppet-collections>.



puppetserver

The `puppetserver` uses the same design principles as PuppetDB. PuppetDB uses a new framework named Trapperkeeper. Trapperkeeper is written in Clojure and is responsible for managing the HTTP/TLS endpoints that are required to serve as a Puppet master server. More information about Trapperkeeper is available at the project website at <https://github.com/puppetlabs/trapperkeeper>.

Building a Puppet master

To build a split Puppet master configuration, we will first start with an empty machine running an enterprise Linux distribution, such as CentOS, RedHat Enterprise Linux, or Springdale Linux. I will be using Springdale Linux 7 for my example machines. More information on Springdale is available at <https://springdale.math.ias.edu/>. I will start by building a machine named `lb` (load balancer), as my first Puppet master. The `puppetserver` process uses a lot of memory; the `lb` machine needs to have at least 2.5GB of memory to allow the `puppetserver` process to run.



If you are setting up a lab environment where you won't run a large number of nodes, you can reconfigure `puppetserver` to use less memory. More information is available at http://docs.puppetlabs.com/puppetserver/latest/install_from_packages.html#memory-allocation.

To enable the `puppetserver` service on a node, install the Puppet Labs yum repository rpm onto the machine. At the time of writing, the latest release rpm is `puppetlabs-release-pc1-0.9.2-1.el7.noarch.rpm`, which is available from Puppet Labs at http://yum.puppetlabs.com/el/7/PC1/x86_64/puppetlabs-release-pc1-0.9.2-1.el7.noarch.rpm.

This is to be installed using the following yum command:

```
[thomas@lb ~]$ sudo yum install http://yum.puppetlabs.com/el/7/PC1/x86_64/puppetlabs-release-pc1-0.9.2-1.el7.noarch.rpm  
puppetlabs-release-pc1-0.9.2-1.el7.noarch.rpm | 4.1 kB  
00:00:00  
...
```

Installed:

```
puppetlabs-release-pc1.noarch 0:0.9.2-1.el7
```

Complete!

After installing the `puppetlabs-release-pc1` rpm, install the `puppetserver` rpm. This can be done with the following command:

```
[thomas@lb ~]$ sudo yum install puppetserver
```

Installing `puppetserver` will automatically install a few Java dependencies. Installing `puppetserver` will also install the `puppet-agent` rpm onto your system. This places the Puppet and Facter applications into `/opt/puppetlabs/bin`. This path may not be in your `PATH` environment variable, so you need to add this to your `PATH` variable either by adding a script to the `/etc/profile.d` directory or appending the path to your shell initialization files.



If you are using `sudo`, then you will have to add `/opt/puppetlabs/bin` to your `secure_path` setting in `/etc/sudoers`, as well.

Now that the server is installed, we'll need to generate new X.509 certificates for our Puppet infrastructure.

Certificates

To generate certificates, we need to initialize a new CA on the `lb` machine. This can be done easily using the `puppet cert` subcommand, as shown here:

```
[thomas@lb ~]$ sudo /opt/puppetlabs/bin/puppet cert list -a
Notice: Signed certificate request for ca
```

With the CA certificate generated, we can now create a new certificate for the master. When nodes connect to Puppet, they will search for a machine named `puppet`. Since the name of my test machine is `lb`, I will alter Puppet configuration to have Puppet believe that the name of the machine is `puppet`. This is done by adding the following to the `puppet.conf` file in either the `[main]` or `[master]` sections. The file is located in `/etc/puppetlabs/puppet/puppet.conf`:

```
certname = puppet.example.com
```

The domain of my test machine is `example.com` and I will generate the certificate for `lb` with the `example.com` domain defined. To generate this new certificate, we will use the `puppet certificate generate` subcommand, as shown here:

```
[thomas@lb ~]$ sudo /opt/puppetlabs/bin/puppet certificate generate
--dns-alt-names puppet,puppet.example.com,puppet.dev.example.com puppet.
example.com --ca-location local
Notice: puppet.example.com has a waiting certificate request
true
```

Now, since the certificate has been generated, we need to sign the certificate, as shown here:

```
[thomas@lb ~]$ sudo /opt/puppetlabs/bin/puppet cert sign puppet.example.
com --allow-dns-alt-names
Notice: Signed certificate request for puppet.example.com
Notice: Removing file Puppet::SSL::CertificateRequest'puppet.example.com'
at '/etc/puppetlabs/puppet/ssl/ca/requests/puppet.example.com.pem'
```

The signed certificate will be placed into the `/etc/puppetlabs/puppet/ssl/ca/signed` directory; we need to place the certificate in the `/etc/puppetlabs/puppet/ssl/certs` directory. This can be done with the `puppet certificate find` command, as shown here:

```
[thomas@lb ~]$ sudo puppet certificate find puppet.example.com --ca-
location local
-----BEGIN CERTIFICATE-----
MIIFvDCCA6SgAwIBAgIBAjANBgkqhkiG9w0BAQsFADAOwMSYwJAYDVQQDDB1QdXBw
```

```
...
9ZLNFwdQ4iMxenffcEQErMfkT6fjcvdSIjShoIe3Myk=
-----END CERTIFICATE-----
```

In addition to displaying the certificate, the `puppet cert sign` command will also place the certificate into the correct directory.

With the certificate in place, we are ready to start the `puppetserver` process.

systemd

Enterprise Linux 7 (EL7) based distributions now use `systemd` to control the starting and stopping of processes. EL7 distributions still support the `service` command to start and stop services. However, using the equivalent `systemd` commands is the preferred method and will be used in this book. `systemd` is a complete rewrite of the System V init process and includes many changes from traditional UNIX init systems. More information on `systemd` can be found on the freedesktop website at <http://www.freedesktop.org/wiki/Software/systemd/>.

To start the `puppetserver` service using `systemd`, use the `systemctl` command, as shown here:

```
[thomas@lb ~]$ sudo systemctl start puppetserver
```

`puppetserver` will start after a lengthy process of creating JVMs. To verify that `puppetserver` is running, verify that the Puppet master port (TCP port 8140) is listening for connections with the following command:

```
[thomas@lb ~]$ sudo lsof -i :8140
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
java 4299 puppet 28u IPv6 37899 0t0 TCP *:8140 (LISTEN)
```

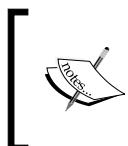
At this point, your server will be ready to accept connections from Puppet agents. To ensure that the `puppetserver` service is started when our machine is rebooted, use the `enable` option with `systemctl`, as shown here:

```
[root@puppet ~]# sudo systemctl enable puppetserver.service
ln -s '/usr/lib/systemd/system/puppetserver.service' '/etc/systemd/
system/multi-user.target.wants/puppetserver.service'
```

With Puppet master running, we can now begin to configure a load balancer for our workload.

Creating a load balancer

At this point, the 1b machine is acting as a Puppet master running the `puppetserver` service. Puppet agents will not be able to connect to this service. By default, EL7 machines are configured with a firewall service that will prevent access to port 8140. At this point, you can either configure the firewall using `firewalld` to allow the connection, or disable the firewall.



Host based firewalls can be useful; by disabling the firewall, any service that is started on our server will be accessible from outside machines. This may potentially expose services we do not wish to expose from our server.



To disable the firewall, issue the following commands:

```
[thomas@client ~]$ sudo systemctl disable firewalld.service
rm '/etc/systemd/system/dbus-org.fedoraproject.FirewallD1.service'
rm '/etc/systemd/system/basic.target.wants/firewalld.service'
[thomas@client ~]$ sudo systemctl stop firewalld.service
```

Alternatively, to allow access to port 8140, issue the following commands:

```
[thomas@lb ~]$ sudo firewall-cmd --add-port=8140/tcp
success
[thomas@lb ~]$ sudo firewall-cmd --add-port=8140/tcp --permanent
success
```

We will now create a load balancing configuration with three servers: our first 1b machine and two machines running `puppetserver` and acting as Puppet masters. I will name these `puppetmaster1` and `puppetmaster2`.

To configure the 1b machine as a load balancer, we need to reconfigure `puppetserver` in order to listen on an alternate port. We will configure Apache to listen on the default Puppet master port of 8140. To make this change, edit the `webserver.conf` file in the `/etc/puppetlabs/puppetserver/conf.d` directory, so that its contents are the following:

```
webserver: {
  access-log-config = /etc/puppetlabs/puppetserver/request-logging.xml
  client-auth = want
  ssl-host = 0.0.0.0
  ssl-port = 8141
  host = 0.0.0.0
  port = 18140
}
```

This will configure `puppetserver` to listen on port 8141 for TLS encrypted traffic and port 18140 for unencrypted traffic. After making this change, we need to restart the `puppetserver` service using `systemctl`, as follows:

```
[thomas@lb ~]$ sudo systemctl restart puppetserver.service
```

Next, we will configure Apache to listen on the master port and act as a proxy to the `puppetserver` process.

Apache proxy

To configure Apache to act as a proxy service for our load balancer, we will need to install `httpd`, the Apache server. We will also need to install the `mod_ssl` package to support encryption on our load balancer. To install both these packages, issue the following yum command:

```
[thomas@lb ~]$ sudo yum install httpd mod_ssl
```

Next, create a configuration file for the load balancer that uses the `puppet.example.com` certificates, which we created earlier. Create a file named `puppet_lb.conf` in the `/etc/httpd/conf.d` directory with the following contents:

```
Listen 8140
<VirtualHost *:8140>
    ServerName puppet.example.com
    SSLEngine on
    SSLProtocol -ALL +TLSv1 +TLSv1.1 +TLSv1.2
    SSLCipherSuite ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:-LOW:-SSLv2:-EXP
    SSLCertificateFile /etc/puppetlabs/puppet/ssl/certs/puppet.example.com.pem
    SSLCertificateKeyFile /etc/puppetlabs/puppet/ssl/private_keys/puppet.example.com.pem
    SSLCertificateChainFile /etc/puppetlabs/puppet/ssl/ca/ca_crt.pem
    SSLCACertificateFile /etc/puppetlabs/puppet/ssl/ca/ca_crt.pem
    # If Apache complains about invalid signatures on the CRL, you can
    # try disabling
    # CRL checking by commenting the next line, but this is not
    # recommended.
    SSLCARevocationFile /etc/puppetlabs/puppet/ssl/ca/ca_crl.pem
    SSLVerifyClient optional
    SSLVerifyDepth 1
    # The `ExportCertData` option is needed for agent certificate
    # expiration warnings
    SSLOptions +StdEnvVars +ExportCertData
    # This header needs to be set if using a loadbalancer or proxy
    RequestHeader unset X-Forwarded-For
```

```
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

ProxyPassMatch ^/(puppet-ca/v[123] /.*$ balancer://puppetca/$1
ProxyPass / balancer://puppetworker/
ProxyPassReverse / balancer://puppetworker

<Proxy balancer://puppetca>
    BalancerMember http://127.0.0.1:18140
</Proxy>
<Proxy balancer://puppetworker>
    BalancerMember http://192.168.0.100:18140
    BalancerMember http://192.168.0.101:18140
</Proxy>

</VirtualHost>
```

This configuration creates an Apache `VirtualHost` that will listen for connections on port 8140 and redirect traffic to one of the three `puppetserver` instances. One `puppetserver` instance is the instance running on the load balancer machine 1b. The other two are Puppet master servers, which we have not built yet. To continue with our configuration, create two new machines and install `puppetserver`, as we did on the 1b machine; name these servers, as `puppetmaster1` and `puppetmaster2`.

In our load balancing configuration, communication between the 1b machine and the Puppet masters will be unencrypted. To maintain security, a private network should be established between the 1b machine and the Puppet masters. In my configuration, I gave the two Puppet masters IP addresses 192.168.0.100 and 192.168.0.101, respectively. The 1b machine was given the IP address 192.168.0.110.

The following lines in the Apache configuration are used to create two proxy balancer locations, using Apache's built-in proxying engine:

```
<Proxy balancer://puppetca>
    BalancerMember http://127.0.0.1:18140
</Proxy>
<Proxy balancer://puppetworker>
    BalancerMember http://192.168.0.100:18140
    BalancerMember http://192.168.0.101:18140
</Proxy>
```

The `puppetca` balancer points to the local `puppetserver` running on 1b. The `puppetworker` balancer points to both `puppetmaster1` and `puppetmaster2` and will round robin between the two machines.

The following `ProxyPass` and `ProxyPassMatch` configuration lines direct traffic between the two balancer endpoints:

```
ProxyPassMatch ^/(puppet-ca/v[123]/*)$ balancer://puppetca/$1  
ProxyPass / balancer://puppetworker/  
ProxyPassReverse / balancer://puppetworker
```

These lines take advantage of the API redesign in Puppet 4. In previous versions of Puppet, the Puppet REST API defined the endpoints using the following syntax:

```
environment/endpoint/value
```

The first part of the path is the environment used by the node. The second part is the endpoint. The endpoint may be one of certificate, file, or catalog (there are other endpoints, but these are the important ones here). All traffic concerned with certificate signing and retrieval will have the word "certificate" as the endpoint. To redirect all certificate related traffic to a specific machine, the following `ProxyPassMatch` directive can be used:

```
ProxyPassMatch ^/([^\/]+/certificate.*)$ balancer://puppetca/$1
```

Indeed, this was the `ProxyPassMatch` line that I used when working with Puppet 3 in the previous version of this book. Starting with Puppet 4, the REST API URLs have been changed, such that all certificate or **certificate authority (CA)** traffic is directed to the `puppet-ca` endpoint. In Puppet 4, the API endpoints are defined, as follows:

```
/puppet-ca/version/endpoint/value?environment=environment
```

Or, as follows:

```
puppet/version/endpoint/value?environment=environment
```

The environment is now placed as an argument to the URL after ?. All CA related traffic is directed to the `/puppet-ca` URL and everything else to the `/puppet` URL.

To take advantage of this, we use the following `ProxyPassMatch` directive:

```
ProxyPassMatch ^/(puppet-ca/v[123]/*)$ balancer://puppetca/$1
```

With this configuration in place, all certificate traffic is directed to the `puppetca` balancer.

In the next section, we will discuss how TLS encryption information is handled by our load balancer.

TLS headers

When a Puppet agent connects to a Puppet master, the communication is authenticated with X.509 certificates. In our load balancing configuration, we are interjecting ourselves between the nodes and the puppetserver processes on the Puppet master servers. To allow the TLS communication to flow, we configure Apache to place the TLS information into headers, as shown in the following configuration lines:

```
# This header needs to be set if using a loadbalancer or proxy
RequestHeader unset X-Forwarded-For
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 lines take information from the connecting nodes and place them into HTTP headers that are then passed to the puppetserver processes. We can now start Apache and begin answering requests on port 8140.

SELinux

Security-Enhanced Linux (SELinux) is a system for Linux that provides support for **mandatory access controls (MAC)**. If your servers are running with SELinux enabled, great! You will need to enable an SELinux Boolean to allow Apache to connect to the puppetserver servers on port 18140. This Boolean is `httpd_can_network_connect`. To set this Boolean, use the `setsebool` command, as shown here:

```
[thomas@lb ~]$ sudo setsebool -P httpd_can_network_connect=1
```

SELinux provides an extra level of security. For this load balancer configuration, the Boolean is the only SELinux configuration change that was required. If you have unexplained errors, you can check for SELinux AVC messages in `/var/log/audit/audit.log`. To allow any access that SELinux is denying, you use the `setenforce` command, as shown here:

```
[thomas@lb ~]$ sudo setenforce 0
```

More information on SELinux is available at http://selinuxproject.org/page/Main_Page.

Now a configuration change must be made for the puppetserver processes to access certificate information passed in headers. The `master.conf` file must be created in the `/etc/puppetlabs/puppetserver/conf.d` directory with the following content:

```
master: {
    allow-header-cert-info: true
}
```

After making this change, `puppetserver` must be restarted.

At this point, there will be three `puppetserver` processes running; there will be one on each of the Puppet masters and another on the `lb` machine.

Before we can use the new master servers, we need to copy the certificate information from the `lb` machine. The quickest way to do this is to copy the entire `/etc/puppetlabs/puppet/ssl` directory to the masters. I did this by creating a TAR file of the directory and copying the TAR file using the following commands:

```
[root@lb puppet]# cd /etc/puppetlabs/puppet  
[root@lb puppet]# tar cf ssl.tar ssl
```

With the certificates in place, the next step is to configure Puppet on the Puppet masters.

Configuring masters

To test the configuration of the load balancer, create `site.pp` manifests in the code production directory `/etc/puppetlabs/code/environments/production/manifests` with the following content:

```
node default {  
    notify { "compiled on puppetmaster1": }  
}
```

Create the corresponding file on `puppetmaster2`:

```
node default {  
    notify { "compiled on puppetmaster2": }  
}
```

With these files in place and the `puppetserver` processes running on all three machines, we can now test our infrastructure. You can begin by creating a client node and installing the `puppetlabs` release package and then the `puppet-agent` package. With Puppet installed, you will need to either configure DNS, such that the `lb` machine is known as `puppet` or add the IP address of the `lb` machine to `/etc/hosts` as the `puppet` machine, as shown here:

```
192.168.0.110 puppet.example.com puppet
```

Next, start the Puppet agent on the client machine. This will create a certificate for the machine on the `lb` machine, as shown here:

```
[thomas@client ~]$ sudo puppet agent -t  
Info: Creating a new SSL key for client
```

```
Info: csr_attributes file loading from /etc/puppetlabs/puppet/csr_attributes.yaml
Info: Creating a new SSL certificate request for client
Info: Certificate Request fingerprint (SHA256): FE:D1:6D:70:90:10:9E:C9:0E:D7:3B:BA:3D:2C:71:93:59:40:02:64:0C:FC:D4:DD:8E:92:EF:02:7F:EE:28:52
Exiting; no certificate found and waitforcert is disabled
```

On the lb machine, list the unsigned certificates with the puppet cert list command, as shown here:

```
[thomas@lb ~]$ sudo puppet cert list
"client" (SHA256) FE:D1:6D:70:90:10:9E:C9:0E:D7:3B:BA:3D:2C:71:93:59:40:02:64:0C:FC:D4:DD:8E:92:EF:02:7F:EE:28:52
```

Now sign the certificate using the puppet cert sign command, as shown:

```
[thomas@lb ~]$ sudo puppet cert sign client
Notice: Signed certificate request for client
Notice: Removing file Puppet::SSL::CertificateRequest client at '/etc/puppetlabs/puppet/ssl/ca/requests/client.pem'
```

With the certificate signed, we can run puppet agent again on the client machine and verify the output:

```
[thomas@client ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client
Info: Applying configuration version '1441254717'
Notice: compiled on puppetserver1
Notice: /Stage[main]/Main/Node[default]/Notify[compiled on puppetmaster1]/message: defined 'message' as 'compiled on puppetmaster1'
Notice: Applied catalog in 0.04 seconds
```

If we run the agent again, we might see another message from the other Puppet master:

```
[thomas@client ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client
Info: Applying configuration version '1441256532'
Notice: compiled on puppetmaster2
```

```
Notice: /Stage[main]/Main/Node[default]/Notify[compiled on
puppetmaster2]/message: defined 'message' as 'compiled on puppetmaster2'
Notice: Applied catalog in 0.02 seconds
```

An important thing to note here is that the certificate for our client machine is only available on the 1b machine. When we list all the certificates available on puppetmaster1, we only see the puppet.localdomain certificate, as shown in the following output:

```
[thomas@puppet ~]$ sudo puppet cert list -a
+ "puppet.example.com" (SHA256) 9B:C8:43:46:71:1E:0A:E0:63:E8:A7:B5:C2
:BF:4D:6E:68:4C:67:57:87:4C:7A:77:08:FC:5A:A6:62:E9:13:2E (alt names:
"DNS:puppet", "DNS:puppet.dev.example.com", "DNS:puppet.example.com")
```

However, running the same command on the 1b machine returns the certificate we were expecting:

```
[thomas@lb ~]$ sudo puppet cert list -a
+ "client" (SHA256) E6:38:60:C9:78:F8:B1:88:EF:3C:58:17:88:81
:72:86:1B:05:C4:00:B2:A2:99:CD:E1:FE:37:F2:36:6E:8E:8B
+ "puppet.example.com" (SHA256) 9B:C8:43:46:71:1E:0A:E0:63:E8:A7:B5:C2
:BF:4D:6E:68:4C:67:57:87:4C:7A:77:08:FC:5A:A6:62:E9:13:2E (alt names:
"DNS:puppet", "DNS:puppet.dev.example.com", "DNS:puppet.example.com")
```

So at this point, when the nodes connect to our 1b machine, all the certificate traffic is directed to the puppetserver process running locally on the 1b machine. The catalog requests will be shared between puppetmaster1 and puppetmaster2, using the Apache proxy module. We now have a load balancing puppet infrastructure. To scale out by adding more Puppet masters, we only need to add them to the proxy balancer in the Apache configuration. In the next section, we'll discuss how to keep the code on the various machines up to date.

Keeping the code consistent

At this point, we are can scale out our catalog compilation to as many servers as we need. However, we've neglected one important thing: we need to make sure that Puppet code on all the workers remains in sync. There are a few ways in which we can do this and when we cover integration with Git in *Chapter 3, Git and Environments*, we will see how to use Git to distribute the code.

rsync

A simple method to distribute the code is with rsync. This isn't the best solution, but for example, you will need to run rsync whenever you change the code. This will require changing the Puppet user's shell from /sbin/nologin to /bin/bash or /bin/rbash, which is a potential security risk.



If your Puppet code is on a filesystem that supports ACLs, then creating an rsync user and giving that user the rights to specific directories within that filesystem is a better option. Using `setfacl`, it is possible to grant write access to the filesystem for a user other than Puppet. For more information on ACLs on Enterprise Linux, visit the Red Hat documentation page at https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Storage_Administration_Guide/ch-acls.html.

First, we create an SSH key for rsync to use to SSH between the Puppet master nodes and the load balancer. We then copy the key into the `authorized_keys` file of the Puppet user on the Puppet masters, using the `ssh-copy-id` command. We start by generating the certificate on the load balancer, as shown here:

```
lb# ssh-keygen -f puppet_rsync
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in puppet_rsync.
Your public key has been saved in puppet_rsync.pub.
```

This creates `puppet_rsync.pub` and `puppet_rsync`. Now, on the Puppet masters, configure the Puppet user on those machines to allow access using this key using the following commands:

```
[thomas@puppet ~]$ sudo mkdir ~puppet/.ssh
[thomas@puppet ~]$ sudo cp puppet_rsync.pub ~puppet/.ssh/authorized_keys
[thomas@puppet ~]$ sudo chown -R puppet:puppet ~puppet/.ssh
[thomas@puppet ~]$ sudo chmod 750 ~puppet
[thomas@puppet ~]$ sudo chmod 700 ~puppet/.ssh
[thomas@puppet ~]$ sudo chmod 600 ~puppet/.ssh/authorized_keys
[thomas@puppet ~]$ sudo chsh -s /bin/bash puppet
Changing shell for puppet.
Shell changed.
[thomas@puppet ~]$ sudo chown -R puppet:puppet /etc/puppetlabs/code
```

The changes made here allow us to access the Puppet master server from the load balancer machine, using the SSH key. We can now use `rsync` to copy our code from the load balancer machine to each of the Puppet masters, as shown here:

```
[thomas@lb ~]$ rsync -e 'ssh -i puppet_rsync' -az /etc/puppetlabs/code/  
puppet@puppetmaster1:/etc/puppetlabs/code
```

 [**Creating SSH keys and using rsync**]

The trailing slash in the first part `/etc/puppetlabs/code/` and the absence of the slash in the second part `puppet@puppetmaster1:/etc/puppetlabs/code` is by design. In this manner, we get the contents of `/etc/puppetlabs/code` on the load balancer placed into `/etc/puppetlabs/code` on the Puppet master.

Using `rsync` is not a good enterprise solution. The concept of using the SSH keys and transferring the files as the Puppet user is useful. In *Chapter 2, Organizing Your Nodes and Data*, we will use this same concept when triggering code updates via Git.

NFS

A second option to keep the code consistent is to use NFS. If you already have an NAS appliance, then using the NAS to share Puppet code may be the simplest solution. If not, using Puppet master as an NFS server is another. However, this makes your Puppet master a big, single point of failure. NFS is not the best solution for this sort of problem.

Clustered filesystem

Using a clustered filesystem, such as `gfs2` or `glusterfs` is a good way to maintain consistency between nodes. This also removes the problem of the single point of failure with NFS. A cluster of three machines makes it far less likely that the failure of a single machine will render the Puppet code unavailable.

Git

The third option is to have your version control system keep the files in sync with a post-commit hook or scripts that call Git directly such as `r10k` or `puppet-sync`. We will cover how to configure Git to do some housekeeping for us in *Chapter 2, Organizing Your Nodes and Data*. Using Git to distribute the code is a popular solution, since it only updates the code when a commit is made. This is the continuous delivery model. If your organization would rather push code at certain points (not automatically), then I would suggest using the scripts mentioned earlier on a routine basis.

One more split

Now that we have our Puppet infrastructure running on two Puppet masters and the load balancer, you might notice that the load balancer and the certificate signing machine need not be the same machine.

To split off the Puppet certificate authority (`puppetca`) from the load balancing machine, make another Puppet master machine, similar to the previous Puppet master machines (complete with the `master.conf` configuration file in the `/etc/puppetlabs/puppetserver/conf.d` directory). Give this new machine the following IP address 192.168.0.111.

Now, modify the `puppet_lb.conf` file in the `/etc/httpd/conf.d` directory such that the proxy balancer for `puppetca` points to this new machine, as shown here:

```
<Proxy balancer://puppetca>
    BalancerMember http://192.168.0.111:18140
</Proxy>
```

Now restart Apache on the load balancer and verify that the certificate signing is now taking place on the new `puppetca` machine. This can be done by running Puppet on our client machine with the `--certname` option to specify an alternate name for our node, as shown here:

```
[thomas@client ~]$ puppet agent -t --certname split
Info: Creating a new SSL key for split
Info: csr_attributes file loading from /home/thomas/.puppetlabs/etc/
puppet/csr_attributes.yaml
Info: Creating a new SSL certificate request for split
Info: Certificate Request fingerprint (SHA256): 98:41:F6:7C:44:FE:35:E5:B
9:B5:86:87:A1:BE:3A:FD:4A:D4:50:B8:3A:3A:69:00:87:12:0D:9A:2B:B0:94:CF
Exiting; no certificate found and waitforcert is disabled
```

Now on the `puppetca` machine, run the `puppet cert list` command to see the certificate waiting to be signed:

```
[thomas@puppet ~]$ sudo puppet cert list
"split" (SHA256) 98:41:F6:7C:44:FE:35:E5:B9:B5:86:87:A1:BE:3A:FD:4A:D4:
50:B8:3A:3A:69:00:87:12:0D:9A:2B:B0:94:CF
```

When we run the `puppet cert list` command on the load balancer, we see that the split certificate is not shown:

```
thomas@lb ~]$sudo puppet cert list -a
+ "client"          (SHA256) E6:38:60:C9:78:F8:B1:88:EF:3C:58:17:88:81
:72:86:1B:05:C4:00:B2:A2:99:CD:E1:FE:37:F2:36:6E:8E:8B
+ "puppet.example.com" (SHA256) 9B:C8:43:46:71:1E:0A:E0:63:E8:A7:B5:C2
:BF:4D:6E:68:4C:67:57:87:4C:7A:77:08:FC:5A:A6:62:E9:13:2E (alt names:
"DNS:puppet", "DNS:puppet.dev.example.com", "DNS:puppet.example.com")
```

With this split we have streamlined the load balancer to the point where it is only running Apache. In the next section, we'll look at how else we can split up our workload.

One last split or maybe a few more

We have already split our workload into a certificate-signing machine (`puppetca`) and a pool of catalog compiling machines (Puppet masters). We can also create a report processing machine and split-off report processing to that machine with the `report_server` setting. What is interesting as an exercise at this point is that we can also serve up files using our load balancing machine.

Based on what we know about the Puppet HTTP API, we know that requests for `file_buckets` and files have specific URIs, which we can serve directly from the load balancer without using `puppetserver`, or Apache or even Puppet. To test the configuration, alter the definition of the default node to include a file, as follows:

```
node default {
  include file_example
}
```

Create the `file_example` module and the following class manifest:

```
class file_example {
  file {'/tmp/example':
    mode=>'644',
    owner =>'100',
    group =>'100',
    source => 'puppet:///modules/file_example/example',
  }
}
```

Create the example file in the `files` subdirectory of the module. In this file, place the following content:

```
This file is in the code directory.
```

Now, we need to edit the Apache configuration on the load balancer to redirect file requests to another VirtualHost on the load balancer. Modify the `puppet_lb.conf` file so that the rewrite balancer lines are, as follows:

```
ProxyPassMatch ^/(puppet-ca/v[123]/.*)$ balancer://puppetca/$1
ProxyPassMatch ^/puppet/v[123]/file_content/(.*)$ balancer://
puppetfile/$1

ProxyPass / balancer://puppetworker/
ProxyPassReverse / balancer://puppetworker

<Proxy balancer://puppetca>
    BalancerMember http://192.168.0.111:18140
</Proxy>
<Proxy balancer://puppetfile>
    BalancerMember http://127.0.0.1:8080
</Proxy>
<Proxy balancer://puppetworker>
    BalancerMember http://192.168.0.100:18140
    BalancerMember http://192.168.0.101:18140
</Proxy>
```

This configuration will redirect any requests to `/puppet/v3/file_content` to port 8080 on the same machine. We now need to configure Apache to listen on port 8080, create the `files.conf` file in the `/etc/httpd/conf.d` directory:

```
Listen 8080
<VirtualHost *:8080>
    DocumentRoot /var/www/html/puppet
    LogLevel debug
    RewriteEngine on
    RewriteCond %{QUERY_STRING}      ^environment=(.*).*$      [NC]
    RewriteRule^(.*)$      /%1/$1      [NC,L]
</VirtualHost>
```

In version 4 of Puppet, the environment is encoded as a parameter to the request URL. The URL requested by the node for the example file is `/puppet/v3/file_content/modules/file_example/example?environment=production&`. The `files.conf` configuration's `RewriteCond` line will capture the environment `production` into `%1`. The `RewriteRule` line will take the requested URL and rewrite it into `/production/modules/file_example/example`. To ensure that the file is available, create the following directory on the load balancer machine:

```
/var/www/html/puppet/production/modules/file_example
```

Create the example file in this directory with the following content:

```
This came from the load balancer
```

Now, restart the Apache process on the load balancer. At this point we can run the Puppet agent on the client node to have the /tmp/example file created on the client node, as shown here:

```
[thomas@client ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client
Info: Applying configuration version '1441338020'
Notice: compiled on puppetmaster1 -- does that work?
Notice: /Stage[main]/Main/Node[default]/Notify[compiled on puppetmaster1
-- does that work?]/message: defined 'message' as 'compiled on
puppetmaster1 -- does that work?'
Notice: /Stage[main]/File_example/File[/tmp/example]/content:

Info: Computing checksum on file /tmp/example
Info: /Stage[main]/File_example/File[/tmp/example]: Filebucketed /tmp/
example to puppet with sum accaac1654696edf141baeeab9d15198
Notice: /Stage[main]/File_example/File[/tmp/example]/content: content
changed '{md5}accaac1654696edf141baeeab9d15198' to '{md5}1a7b177fb5017e17
daf9522e741b2f9b'
Notice: Applied catalog in 0.23 seconds
[thomas@client ~]$ cat /tmp/example
This came from the load balancer
```

The contents of the file have now been placed on the client machine and, as we can see, the contents of the file are coming from the file that is in the subdirectory of /var/www/html.



One important thing to be considered is security, as any configured client can retrieve files from our gateway machine. In production, you might want to add ACLs to the file location.

As we have seen, once the basic proxying is configured, further splitting up of the workload becomes a routine task. We can split the workload to scale to handle as many nodes as we require.

Conquer by dividing

Depending on the size of your deployment and the way you connect to all your nodes, a masterless solution may be a good fit. In a masterless configuration, you don't run the Puppet agent; rather, you push Puppet code to a node and then run the `puppet apply` command. There are a few benefits to this method and a few drawbacks, as stated in the following table:

Benefits	Drawbacks
No single point of failure	Can't use built-in reporting tools, such as dashboard
Simpler configuration	Exported resources require nodes having write access to the database.
Finer-grained control on where the code is deployed	Each node has access to all the code
Multiple simultaneous runs do not affect each other (reduces contention)	More difficult to know when a node is failing to apply a catalog correctly
Connection to Puppet master not required (offline possible)	No certificate management
No certificate management	

The idea with a masterless configuration is that you distribute Puppet code to each node individually and then kick off a Puppet run to apply that code. One of the benefits of Puppet is that it keeps your system in a good state; so when choosing masterless, it is important to build your solution with this in mind. A cron job configured by your deployment mechanism that can apply Puppet to the node on a routine schedule will suffice.

The key parts of a masterless configuration are: distributing the code, pushing updates to the code, and ensuring that the code is applied routinely to the nodes. Pushing a bunch of files to a machine is best done with some sort of package management.

Many masterless configurations use Git to have clients pull the files, this has the advantage of clients pulling changes. For Linux systems, the big players are rpm and dpkg, whereas for Mac OS, installer package files can be used. It is also possible to configure the nodes to download the code themselves from a web location. Some large installations use Git to update the code, as well.

The solution I will outline is that of using an rpm deployed through yum to install and run Puppet on a node. Once deployed, we can have the nodes pull updated code from a central repository rather than rebuild the rpm for every change.

Creating an rpm

To start our rpm, we will make an rpm spec file. We can make this anywhere since we don't have a master in this example. Start by installing `rpm-build`, which will allow us to build the rpm.

```
# yum install rpm-build  
Installing  
rpm-build-4.8.0-37.el6.x86_64
```

Later, it is important to have a user to manage the repository, so create a user called `builder` at this point. We'll do this on the Puppet master machine we built earlier. Create an `rpmbuild` directory with the appropriate subdirectories and then create our example code in this location:

```
# sudo -iu builder  
$ mkdir -p rpmbuild/{SPECS,SOURCES}  
$ cd SOURCES  
$ mkdir -p modules/example/manifests  
$ cat <<EOF> modules/example/manifests/init.pp  
class example {  
    notify {"This is an example.": }  
    file {'/tmp/example':  
        mode => '0644',  
        owner => '0',  
        group => '0',  
        content => 'This is also an example.'  
    }  
}  
EOF  
$ tar cjf example.com-puppet-1.0.tar.bz2 modules
```

Next, create a spec file for our rpm in `rpmbuild/SPECS` as shown here:

```
Name:           example.com-puppet  
Version: 1.0  
Release: 1%{?dist}  
Summary: Puppet Apply for example.com
```

```
Group: System/Utilities
License: GNU
Source0: example.com-puppet-%{version}.tar.bz2
BuildRoot: %(mktemp -ud %{_tmppath}/%{name}-%{version}-%{release}-XXXXXX)

Requires: puppet
BuildArch: noarch

%description
This package installs example.com's puppet configuration
and applies that configuration on the machine.

%prep

%setup -q -c
%install
mkdir -p $RPM_BUILD_ROOT/%{_localstatedir}/local/puppet
cp -a . $RPM_BUILD_ROOT/%{_localstatedir}/local/puppet

%clean
rm -rf %{buildroot}

%files
%defattr(-,root,root,-)
%{_localstatedir}/local/puppet

%post
# run puppet apply
/bin/env puppet apply --logdest syslog --modulepath=%{_localstatedir}/
local/puppet/modules %{_localstatedir}/local/puppet/manifests/site.pp

%changelog
* Fri Dec 6 2013 Thomas Uphill <thomas@narrabilis.com> - 1.0-1
- initial build
```

Then use the `rpmbuild` command to build the rpm based on this spec, as shown here:

```
$ rpmbuild -baexample.com-puppet.spec
...
Wrote: /home/builder/rpmbuild/SRPMS/example.com-puppet-1.0-1.el6.src.rpm
Wrote: /home/builder/rpmbuild/RPMS/noarch/example.com-puppet-1.0-1.el6.noarch.rpm
```

Now, deploy a node and copy the rpm onto that node. Verify that the node installs Puppet and then does a Puppet apply run.

```
# yum install example.com-puppet-1.0-1.el6.noarch.rpm
Loaded plugins: downloadonly
...
Installed:
example.com-puppet.noarch 0:1.0-1.el6
Dependency Installed:
augeas-libs.x86_64 0:1.0.0-5.el6
...
puppet-3.3.2-1.el6.noarch
...
Complete!
```

Verify that the file we specified in our package has been created using the following command:

```
# cat /tmp/example
This is also an example.
```

Now, if we are going to rely on this system of pushing Puppet to nodes, we have to make sure that we can update the rpm on the clients and we have to ensure that the nodes still run Puppet regularly, so as to avoid configuration drift (the whole point of Puppet).

Using Puppet resource to configure cron

There are many ways to accomplish these two tasks. We can put the cron definition into the post section of our rpm, as follows:

```
%post
# install cron job
/bin/env puppet resource cron 'example.com-puppet' command='/bin/
env puppet apply --logdest syslog --modulepath=%{_localstatedir}/
```

```
local/puppet/modules %{_localstatedir}/local/puppet/manifests/site.pp'
minute='*/30' ensure='present'
```

We can have a cron job be part of our `site.pp`, as shown here:

```
cron { 'example.com-puppet':
  ensure  => 'present',
  command => '/bin/env puppet apply --logdest syslog --modulepath=/var/local/puppet/modules /var/local/puppet/manifests/site.pp',
  minute   => ['*/30'],
  target   => 'root',
  user     => 'root',
}
```

To ensure that the nodes have the latest versions of the code, we can define our package in `site.pp`:

```
package {'example.com-puppet': ensure => 'latest' }
```

In order for that to work as expected, we need to have a yum repository for the package and have the nodes looking at that repository for packages.

Creating the yum repository

Creating a yum repository is a very straightforward task. Install the `createrepo` rpm and then run `createrepo` on each directory you wish to make into a repository:

```
# mkdir /var/www/html/puppet
# yum install createrepo
...
Installed:
createrepo.noarch 0:0.9.9-18.el6
# chown builder /var/www/html/puppet
# sudo -iu builder
$ mkdir /var/www/html/puppet/{noarch,SRPMS}
$ cp /home/builder/rpmbuild/RPMS/noarch/example.com-puppet-1.0-1.el6.noarch.rpm /var/www/html/puppet/noarch
$ cp rpmbuild/SRPMS/example.com-puppet-1.0-1.el6.src.rpm /var/www/html/puppet/SRPMS
$ cd /var/www/html/puppet
$ createrepo noarch
$ createrepo SRPMS
```

Our repository is ready, but we need to export it with the web server to make it available to our nodes. This rpm contains all our Puppet code, so we need to ensure that only the clients we wish get an access to the files. We'll create a simple listener on port 80 for our Puppet repository:

```
Listen 80
<VirtualHost *:80>
    DocumentRoot /var/www/html/puppet
</VirtualHost>
```

Now, the nodes need to have the repository defined on them so that they can download the updates when they are made available via the repository. The idea here is that we push the rpm to the nodes and have them install the rpm. Once the rpm is installed, the yum repository pointing to updates is defined and the nodes continue updating themselves:

```
yumrepo { 'example.com-puppet':
    baseurl  => 'http://puppet.example.com/noarch',
    descr    => 'example.com Puppet Code Repository',
    enabled   => '1',
    gpgcheck  => '0',
}
```

So, to ensure that our nodes operate properly, we have to make sure of the following things:

1. Install code.
2. Define repository.
3. Define cron job to run Puppet apply routinely.
4. Define package with *latest* tag to ensure it is updated.

A default node in our masterless configuration requires that the cron task and the repository be defined. If you wish to segregate your nodes into different production zones (such as development, production, and sandbox), I would use a repository management system, such as Pulp. Pulp allows you to define repositories based on other repositories and keeps all your repositories consistent.



You should also set up a gpg key on the builder account that can sign the packages it creates. You will then distribute the gpg public key to all your nodes and enable `gpgcheck` on the repository definition.

Downloading the example code

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

You can download the code files by following these steps:



1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

Summary

Dealing with scale is a very important task in enterprise deployments. In the first section, we configured a Puppet master with `puppetserver`. We then expanded the configuration with load balancing and proxying techniques realizing that Puppet is simply a web service. Understanding how nodes request files, catalogs, and certificates allows you to modify the configurations and bypass or alleviate bottlenecks.

In the last section, we explored masterless configuration, wherein instead of checking into Puppet to retrieve new code, the nodes check out the code first and then run against it on a schedule.

Now that we have dealt with the load issue, we need to turn our attention to managing the modules to be applied to nodes. We will cover the organization of the nodes in the next chapter.

2

Organizing Your Nodes and Data

Now that we can deal with a large number of nodes in our installation, we need a way to organize the classes we apply to each node.

There are a few solutions to the problem of attaching classes to nodes. In this chapter, we will examine the following node organization methods:

- An **external node classifier** (ENC)
- LDAP backend
- Hiera

Getting started

For the remainder of this chapter, we will assume that your Puppet infrastructure is configured with a single Puppet master running `puppetserver`. We will name this server `puppet` and give it the IP address `192.168.1.1`. Any Puppet master configuration will be sufficient for this chapter; the configuration from the previous chapter was used in the examples of this chapter.

Organizing the nodes with an ENC

An ENC is a process that is run on the Puppet master or the host compiling the catalog, to determine which classes are applied to the node. The most common form of ENC is a script run through the `exec` node terminus. When using the `exec` node terminus, the script can be written in any language and it receives `certname` (certificate name) from the node, as a command-line argument. In most cases, this will be the **Fully Qualified Domain Name (FQDN)** of the node. We will assume that the `certname` setting has not been explicitly set and that the FQDN of our nodes is being used.

We will only use the hostname portion, as the FQDN can be unreliable in some instances. Across your enterprise, the naming convention of the host should not allow multiple machines to have the same hostname. The FQDN is determined by a fact; this fact is the union of the hostname fact and the domain fact. The domain fact on Linux is determined by running the `hostname -f` command. If DNS is not configured correctly or reverse records do not exist, the domain fact will not be set and the FQDN will also not be set, as shown:

```
# facter domain
example.com
# facter fqdn
node1.example.com
# mv /etc/resolv.conf /etc/resolv.conf.bak
# facter domain
# facter fqdn
#
```

The output of the ENC script is a YAML file that defines the classes, variables, and environment for the node.

Unlike `site.pp`, the ENC script can only assign classes, make top-scope variables, and set the environment of the node. The environment is only set from ENC on versions 3 and above of Puppet.

A simple example

To use an ENC, we need to make one small change in our Puppet master machine. We'll have to add the `node_terminus` and `external_nodes` lines to the `[master]` section of `puppet.conf`, as shown in the following code (we only need make this change on the master machines, as this is concerned with catalog compilation only):

```
[master]
node_terminus = exec
external_nodes = /usr/local/bin/simple_node_classifier
```

 The `puppet.conf` files need not be the same across our installation; Puppet masters and CA machines can have different settings. Having different configuration settings is advantageous in a **Master-of-Master (MoM)** configuration. MoM is a configuration where a top level Puppet master machine is used to provision all of the Puppet master machines.

Our first example, as shown in the following code snippet, will be written in Ruby and live in the file /usr/local/bin/simple_node_classifier, as shown:

```
#!/bin/env ruby
require 'yaml'

# create an empty hash to contain everything
@enc = Hash.new
@enc["classes"] = Hash.new
@enc["classes"]["base"] = Hash.new
@enc["parameters"] = Hash.new
@enc["environment"] = 'production'
#convert the hash to yaml and print
puts @enc.to_yaml
exit(0)
```

Make this script executable and test it on the command line, as shown in the following example:

```
# chmod 755 /usr/local/bin/simple_node_classifier
# /usr/local/bin/simple_node_classifier
---
classes:
  base: {}
environment: production
parameters: {}
```

Puppet version 4 no longer requires the Ruby system package; Ruby is installed in /opt/puppetlabs/puppet/bin. The preceding script relies on Ruby being found in the current \$PATH. If Ruby is not in the current \$PATH, either modify your \$PATH to include /opt/puppetlabs/puppet/bin or install the Ruby system package.

The previous script returns a properly formatted YAML file.

YAML files start with three dashes (---); they use colons (:) to separate parameters from values and hyphens (-) to separate multiple values (arrays). For more information on YAML, visit <http://www.yaml.org/>.

If you use a language such as Ruby or Python, you do not need to know the syntax of YAML, as the built-in libraries take care of the formatting for you. The following is the same example in Python. To use the Python example, you will need to install PyYAML, which is the Python YAML interpreter, using the following command:

```
# yum install PyYAML
Installed:
PyYAML.x86_64 0:3.10-3.el6
```

The Python version starts with an empty dictionary. We then use sub-dictionaries to hold the classes, parameters, and environment. We will call our Python example /usr/local/bin/simple_node_classifier_2. The following is our example:

```
#!/bin/env python
import yaml
import sys
# create an empty hash
enc = {}
enc["classes"] = {}
enc["classes"]["base"] = {}
enc["parameters"] = {}
enc["environment"] = 'production'
# output the ENC as yaml
print "---"
print yaml.dump(enc)
sys.exit(0)
```

Make /usr/local/bin/simple_node_classifier_2 executable and run it using the following commands:

```
worker1# chmod 755 /usr/local/bin/simple_node_classifier_2
worker1# /usr/local/bin/simple_node_classifier_2
---
classes:
  base: {}
environment: production
parameters: {}
```

The order of the lines following --- may be different on your machine; the order is not specified when Python dumps the hash of values.

The Python script outputs the same YAML, as the Ruby code. We will now define the base class referenced in our ENC script, as follows:

```
class base {
  file {'/etc/motd':
    mode     => '0644',
    owner    => '0',
    group   => '0',
    content => inline_template("Managed Node: <%= @hostname %>\n"
                               "Managed by Puppet version <%= @puppetversion %>\n"),
  }
}
```

Now that our base class is defined, modify the `external nodes` setting to point at the Python ENC script. Restart `puppetserver` to ensure that the change is implemented.

Now, run Puppet on the client node. Notice that the message of the day (`/etc/motd`) has been updated using an inline template, as shown in the following command-line output:

```
[thomas@client ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client
Info: Applying configuration version '1441950102'
Notice: /Stage[main]/Base/File[/etc/motd]/ensure: defined content as
'{md5}df3dfe6fe2367e36f0505b486aa24da5'
Notice: Applied catalog in 0.05 seconds
[thomas@client ~]$ cat /etc/motd
Managed Node: client
Managed by Puppet version 4.2.1
```

Since the ENC is only given one piece of data, the `certname` (FQDN), we need to create a naming convention that provides us with enough information to determine the classes that should be applied to the node.

Hostname strategy

In an enterprise, it's important that your hostnames are meaningful. By meaningful, I mean that the hostname should give you as much information as possible about the machine. When you encounter a machine in a large installation, it is likely that you did not build the machine. You need to be able to know as much as possible about the machine just from its name. The following key points should be readily determined from the hostname:

- Operating system
- Application/role
- Location
- Environment
- Instance

It is important that the convention should be standardized and consistent. In our example, let us suppose that the application is the most important component for our organization, so we put that first and the physical location comes next (which data center), followed by the operating system, environment, and instance number. The instance number will be used when you have more than one machine with the same role, location, environment, and operating system. Since we know that the instance number will always be a number, we can omit the underscore between the operating system and environment; thus, making the hostname a little easier to type and remember.

Your enterprise may have more or less information, but the principle will remain the same. To delineate our components, we will use underscores(_). Some companies rely on a fixed length for each component of the hostname, so as to mark the individual components of the hostname by position alone.

In our example, we have the following environments:

- p: This stands for production
- n: This stands for non-production
- d: This stands for development/testing/lab

Our applications will be of the following types:

- web
- db

Our operating system will be Linux, which we will shorten to just l and our location will be our main datacenter (main). So, a production web server on Linux in the main datacenter will have the hostname `web_main_lp01`.



If you think you are going to have more than 99 instances of any single service, you might want to have another leading zero to the instance number (001).

Based only on the hostname, we know that this is a web server in our main datacenter. It's running on Linux and it's the first such machine in production. Now that we have this nice convention, we need to modify our ENC to utilize this convention to glean all the information from the hostname.

Modified ENC using hostname strategy

We'll build our Python ENC script (`/usr/local/bin/simple_node_classifier_2`) and update it to use the new hostname strategy, as follows:

```
#!/bin/env python
# Python ENC
# receives fqdn as argument

import yaml
import sys
"""output_yaml renders the hash as yaml and exits cleanly"""
def output_yaml(enc):
    # output the ENC as yaml
    print "---"
    print yaml.dump(enc)
    quit()
```

Python is very particular about spacing; if you are new to Python, take care to copy the indentations exactly as given in the previous snippet.

We define a function to print the YAML and exit the script. We'll exit the script early if the hostname doesn't match our naming standards, as shown in the following example:

```
# create an empty hash
enc = {}
enc["classes"] = {}
enc["classes"]["base"] = {}
enc["parameters"] = {}

try:
    hostname=sys.argv[1]
except:
    # need a hostname
    sys.exit(10)
```

Exit the script early if the hostname is not defined. This is the minimum requirement and we should never reach this point.

We first split the hostname using underscores (_) into an array called parts and then assign indexes of parts to role, location, os, environment, and instance, as shown in the following code snippet:

```
# split hostname on _
try:
    parts = hostname.split('_')
    role = parts[0]
    location = parts[1]
    os = parts[2][0]
    environment = parts[2][1]
    instance = parts[2][2:]
```

We are expecting hostnames to conform to the standard. If you cannot guarantee this, then you will have to use something similar to the regular expression module to deal with the exceptions to the naming standard:

```
except:
    # hostname didn't conform to our standard
    # include a class which notifies us of the problem
    enc["classes"]["hostname_problem"] = {'enc_hostname': hostname}
    output_yaml(enc)
    raise SystemExit
```

We wrapped the previous assignments in a try statement. In this except statement, we exit printing the YAML and assign a class named hostname_problem. This class will be used to alert us in the console or report to the system that the host has a problem. We send the enc_hostname parameter to the hostname_problem class with the {'enc_hostname': hostname} code.

The environment is a single character in the hostname; hence, we use a dictionary to assign a full name to the environment, as shown here:

```
# map environment from hostname into environment
environments = {}
environments['p'] = 'production'
environments['n'] = 'nonprod'
environments['d'] = 'devel'
environments['s'] = 'sbx'
try:
    enc["environment"] = environments[environment]
except:
    enc["environment"] = 'undef'
```

The following is used to map a role from hostname into role:

```
# map role from hostname into role
enc["classes"][role] = {}
```

Next, we assign top scope variables to the node based on the values we obtained from the parts array previously:

```
# set top scope variables
enc["parameters"]["enc_hostname"] = hostname
enc["parameters"]["role"] = role
enc["parameters"]["location"] = location
enc["parameters"]["os"] = os
enc["parameters"]["instance"] = instance

output_yaml(enc)
```

We will have to define the web class to be able to run the Puppet agent on our `web_main_lp01` machine, as shown in the following code:

```
class web {
  package {'httpd':
    ensure => 'installed'
  }
  service {'httpd':
    ensure  => true,
    enable  => true,
    require => Package['httpd'],
  }
}
```

Heading back to `web_main_lp01`, we run Puppet, sign the certificate on our `puppetca` machine, and then run Puppet again to verify that the `web` class is applied, as shown here:

```
[thomas@web_main_lp01 ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for web_main_lp01.example.com
Info: Applying configuration version '1441951808'
Notice: /Stage[main]/Web/Package[httpd]/ensure: created
Notice: /Stage[main]/Web/Service[httpd]/ensure: ensure changed 'stopped' to 'running'
Info: /Stage[main]/Web/Service[httpd]: Unscheduling refresh on Service[httpd]
Notice: Applied catalog in 16.03 seconds
```

Our machine has been installed as a web server without any intervention on our part. The system knew which classes were to be applied to the machine based solely on the hostname. Now, if we try to run Puppet against our `client` machine created earlier, our ENC will include the `hostname_problem` class with the parameter of the hostname passed to it. We can create this class to capture the problem and notify us. Create the `hostname_problem` module in `/etc/puppet/modules/hostname_problem/manifests/init.pp`, as shown in the following snippet:

```
class hostname_problem ($enc_hostname) {
    notify { "WARNING: $enc_hostname ($::ipaddress) doesn't conform to
naming standards": }
}
```

Now, when we run Puppet on our `node1` machine, we will get a useful warning that `node1` isn't a good hostname for our enterprise, as shown here:

```
[thomas@client ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.example.com
Info: Applying configuration version '1442120036'
Notice: WARNING: client.example.com (10.0.2.15) doesn't conform to naming
standards
Notice: /Stage[main]/Hostname_problem/Notify[WARNING: client.example.
com (10.0.2.15) doesn't conform to naming standards]/message: defined
'message' as 'WARNING: client.example.com (10.0.2.15) doesn't conform to
naming standards'
Notice: Applied catalog in 0.03 seconds
```

Your ENC can be customized much further than this simple example. You have the power of Python, Ruby, or any other language you wish to use. You could connect to a database and run some queries to determine the classes to be installed. For example, if you have a CMDB in your enterprise, you could connect to the CMDB and retrieve information based on the FQDN of the node and apply classes based on that information. You could connect to a URI and retrieve a catalog (dashboard and foreman do something similar). There are many ways to expand this concept.

In the next section, we'll look at using LDAP to store class information.

LDAP backend

If you already have an LDAP implementation in which you can extend the schema, then you can use the LDAP node terminus that is shipped with Puppet. The support for this backend for `puppetserver` has not been maintained as well as it was in the previous releases of Puppet. I still feel that this backend is useful for certain installations. I will outline the steps to be taken to have this backend operate with a `puppetserver` installation. Using this schema adds a new `objectclass` called `puppetclass`. Using this `objectclass`, you can set the environment, set top scope variables, and include classes. The LDAP schema that is shipped with Puppet includes `puppetClass`, `parentNode`, `environment`, and `puppetVar` attributes that are assigned to the `objectclass` named `puppetClient`. The LDAP experts should note that all four of these attributes are marked as optional and the `objectclass` named `puppetClient` is non-structural. To use the LDAP terminus, you must have a working LDAP implementation; apply the Puppet schema to that installation and add the `ruby-ldap` package to your Puppet masters (to allow the master to query for node information).

OpenLDAP configuration

We'll begin by setting up a fresh OpenLDAP implementation and adding a Puppet schema. Create a new machine and install `openldap-servers`. My installation installed the `openldap-servers-2.4.39-6.el7.x86_64` version. This version requires configuration with **OLC (OpenLDAP configuration or runtime configuration)**. Further information on OLC can be obtained at <http://www.openldap.org/doc/admin24/slapdconf2.html>. OLC configures LDAP using LDAP.

After installing `openldap-servers`, your configuration will be in `/etc/openldap/slapd.d/cn=config`. There is a file named `olcDatabase={2}.hdb.ldif` in this directory; edit the file and change the following lines:

```
olcSuffix: dc=example,dc=com
olcRootDN: cn=Manager,dc=example,dc=com
olcRootPW: packtpub
```

Note that the `olcRootPW` line is not present in the default file, so you will have to add it here. If you're going into production with LDAP, you should set `olcDbConfig` parameters as outlined at <http://www.openldap.org/doc/admin24/slapdconf2.html>.

These lines set the top-level location for your LDAP and the password for `RootDN`. This password is in plain text; a production installation would use SSHA encryption. You will be making schema changes, so you must also edit `olcDatabase={0} config.ldif` and set `RootDN` and `RootPW`. For our example, we will use the default `RootDN` value and set the password to `packtpub`, as shown here:

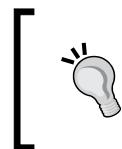
```
olcRootDN: cn=config  
olcRootPW: packtpub
```

These two lines will not exist in the default configuration file provided by the rpm. You might want to keep this `RootDN` value and the previous `RootDN` values separate so that this `RootDN` value is the only one that can modify the schema and top-level configuration parameters.

Next, use `ldapsearch` (provided by the `openldap-clients` package, which has to be installed separately) to verify that LDAP is working properly. Start `slapd` with the `systemctl start slapd.service` command and then verify with the following `ldapsearch` command:

```
# ldapsearch -LLL -x -b'dc=example,dc=com'  
No such object (32)
```

This result indicates that LDAP is running but the directory is empty. To import the Puppet schema into this version of OpenLDAP, copy the `puppet.schema` from <https://github.com/puppetlabs/puppet/blob/master/ext/ldap/puppet.schema> to `/etc/openldap/schema`.



To download the file from the command line directly, use the following command:

```
# curl -O https://raw.githubusercontent.com/  
puppetlabs/puppet/master/ext/ldap/puppet.schema
```

Then create a configuration file named `/tmp/puppet-ldap.conf` with an `include` line pointing to that schema, as shown in the following snippet:

```
include /etc/openldap/schema/puppet.schema
```

Then run `slaptest` against that configuration file, specifying a temporary directory as storage for the configuration files created by `slaptest`, as shown here:

```
# mkdir /tmp/puppet-ldap  
# slaptest -f puppet-ldap.conf -F /tmp/puppet-ldap/  
config file testing succeeded
```

This will create an OLC structure in `/tmp/puppet-ldap`. The file we need is in `/tmp/puppet-ldap/cn=config/cn=schema/cn={0}puppet.ldif`. To import this file into our LDAP instance, we need to remove the ordering information (the braces and numbers `{0}, {1}, ...`) in this file). We also need to set the location for our schema, `cn=schema`, `cn=config`. All the lines after `structuralObjectClass` should be removed. The final version of the file will be in `/tmp/puppet-ldap/cn=config/cn=schema/cn={0}puppet.ldif` and will be as follows:

```
dn: cn=puppet,cn=schema,cn=config
objectClass: olcSchemaConfig
cn: puppet
olcAttributeTypes: ( 1.3.6.1.4.1.34380.1.1.3.10 NAME 'puppetClass'
DESC 'Puppet Node Class' EQUALITY caseIgnoreIA5Match SYNTAX
1.3.6.1.4.1.1466.115.121.1.26 )
olcAttributeTypes: ( 1.3.6.1.4.1.34380.1.1.3.9 NAME 'parentNode'
DESC 'Puppet Parent Node' EQUALITY caseIgnoreIA5Match SYNTAX
1.3.6.1.4.1.1466.115.121.1.26 SINGLE-VALUE )
olcAttributeTypes: ( 1.3.6.1.4.1.34380.1.1.3.11 NAME 'environment'
DESC 'Puppet Node Environment' EQUALITY caseIgnoreIA5Match SYNTAX
1.3.6.1.4.1.1466.115.121.1.26 )
olcAttributeTypes: ( 1.3.6.1.4.1.34380.1.1.3.12 NAME 'puppetVar' DESC
'A variable setting for puppet' EQUALITY caseIgnoreIA5Match SYNTAX
1.3.6.1.4.1.1466.115.121.1.26 )
olcObjectClasses: ( 1.3.6.1.4.1.34380.1.1.1.2 NAME 'puppetClient' DESC
'Puppet Client objectclass' SUP top AUXILIARY MAY ( puppetclass $
parentnode $ environment $ puppetvar ) )
```

Now add this new schema to our instance using `ldapadd`, as follows using the `RootDN` value `cn=config`:

```
# ldapadd -x -f cn=\{0\}puppet.ldif -D 'cn=config' -W
Enter LDAP Password: packtpub
adding new entry "cn=puppet,cn=schema,cn=config"
```

Now we can start adding nodes to our LDAP installation. We'll need to add some containers and a top-level organization to the database before we can do that. Create a file named `start.ldif` with the following contents:

```
dn: dc=example,dc=com
objectclass: dcObject
objectclass: organization
o: Example
dc: example

dn: ou=hosts,dc=example,dc=com
objectclass: organizationalUnit
```

```
ou: hosts

dn: ou=production,ou=hosts,dc=example,dc=com
objectclass: organizationalUnit
ou: production
```

If you are unfamiliar with how LDAP is organized, review the information at http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol#Directory_structure.

Now add the contents of start.ldif to the directory using ldapadd, as follows:

```
# ldapadd -x -f start.ldif -D'cn=manager,dc=example,dc=com' -W
Enter LDAP Password: packtpub
adding new entry "dc=example,dc=com"
adding new entry "ou=hosts,dc=example,dc=com"
adding new entry "ou=production,ou=hosts,dc=example,dc=com"
```

At this point, we have a container for our nodes at ou=production,ou=hosts,dc=example,dc=com. We can add an entry to our LDAP with the following LDIF, which we will name web_main_lp01.ldif:

```
dn: cn=web_main_lp01,ou=production,ou=hosts,dc=example,dc=com
objectclass: puppetClient
objectclass: device
puppetClass: web
puppetClass: base
puppetvar: role='Production Web Server'
```

We then add this LDIF to the directory using ldapadd again, as shown here:

```
# ldapadd -x -f web_main_lp01.ldif -D'cn=manager,dc=example,dc=com' -W
Enter LDAP Password: packtpub
adding new entry "cn=web_main_lp01,ou=production,ou=hosts,dc=example,dc=com"
```

With our entry in LDAP, we are ready to configure our worker nodes to look in LDAP for node definitions. Change /etc/puppetlabs/puppet/puppet.conf to have the following lines in the [master] section:

```
node_terminus = ldap
ldapserver = ldap.example.com
ldapbase = ou=hosts,dc=example,dc=com
```

We are almost ready; `puppetserver` runs Ruby within a Java process. To have this process access our LDAP server, we need to install the `jruby-ldap` gem. `puppetserver` includes a gem installer for this purpose, as shown here:

```
# puppetserver gem install jruby-ldap
Fetching: jruby-ldap-0.0.2.gem (100%)
Successfully installed jruby-ldap-0.0.2
1 gem installed
```

There is a bug in the `jruby-ldap` that we just installed; it was discovered by my colleague Steve Huston on the following Google group: <https://groups.google.com/forum/#!topic/puppet-users/DKu4e7dzhvE>. To patch the `jruby-ldap` module, edit the `conn.rb` file in `/opt/puppetlabs/server/data/puppetserver/jruby-gems/gems/jruby-ldap-0.0.2/lib/ldap` and add the following lines to the beginning:

```
if RUBY_PLATFORM =~ /^java.*$/i
  class LDAP::Entry
    def to_hash
      h = {}
      get_attributes.each { |a| h[a.downcase.to_sym] = self[a] }
      h[:dn] = [dn]
      h
    end
  end
end
```

Restart the `puppetserver` process after making this modification with the `systemctl restart puppetserver.service` command.

 The LDAP backend is clearly not a priority project for Puppet. There are still a few unresolved bugs with using this backend. If you wish to integrate with your LDAP infrastructure, I believe writing your own script that references LDAP will be more stable and easier for you to support.

To convince yourself that the node definition is now coming from LDAP, modify the base class in `/etc/puppet/modules/base/manifests/init.pp` to include the `role` variable, as shown in the following snippet:

```
class base {
  file {'/etc/motd':
    mode => '0644',
    owner => '0',
```

```
group => '0',
  content => inline_template("Role: <%= @role %>\nManaged Node: <%= @hostname %>\nManaged by Puppet version <%= @puppetversion %>\n"),
}
}
```

You will also need to open port 389, the standard LDAP port, on your LDAP server, `ldap.example.com`, to allow Puppet masters to query the LDAP machine.

Then, run Puppet on `web_main_lp01` and verify the contents of `/etc/motd` using the following command:

```
# cat /etc/motd
Role: 'Production Web Server'
Managed Node: web_main_lp01
Managed by Puppet version 4.2.1
```

Keeping your class and variable information in LDAP makes sense if you already have all your nodes in LDAP for other purposes, such as DNS or DHCP. One potential drawback of this is that all the class information for the node has to be stored within a single LDAP entry. It is useful to be able to apply classes to machines based on criteria. In the next section, we will look at Hiera, a system that can be used for this type of criteria-based application.

Before starting the next section, comment out the LDAP ENC lines in `/etc/puppetlabs/puppet/puppet.conf` as follows:

```
# node_terminus = ldap
# ldapserver = puppet.example.com
# ldapbase = ou=hosts,dc=example,dc=com
```

Hiera

Hiera allows you to create a hierarchy of node information. Using Hiera, you can separate your variables and data from your modules. You start by defining what that hierarchy will be, by ordering lookups in the main configuration file, `hiera.yaml`. The hierarchy is based on facts. Any fact can be used, even your own custom facts may be used. The values of the facts are then used as values for the YAML files stored in a directory, usually called `hieradata`. More information on Hiera can be found on the Puppet Labs website at <http://docs.puppetlabs.com/hiera/latest>.



Facts are case sensitive in Hiera and templates. This could be important when writing your `hiera.yaml` script.

Configuring Hiera

Hiera only needs to be installed on your Puppet master nodes. Using the Puppet Labs repo, Hiera is installed by the `puppet-agent` package. Our installation pulled down `puppet-agent-1.2.2-1.el7.x86_64`, which installs Hiera version 3.0.1, as shown here:

```
[thomas@stand ~]$ hiera --version
3.0.1
```

Previous versions of the command-line Hiera tool would expect the Hiera configuration file, `hiera.yaml`, in `/etc/hiera.yaml`. The previous versions of Puppet would expect the configuration file in `/etc/puppet/hiera.yaml` or `/etc/puppetlabs/puppet/hiera.yaml`, depending on the version of Puppet. This caused some confusion, as the command-line utility will, by default, search in a different file than Puppet. This problem has been corrected in Puppet 4; the Hiera configuration file is now located in the `/etc/puppetlabs/code` directory. The default location for the `hieradata` directory includes the value of the current environment and is located at `/etc/puppetlabs/code/environments/%{environment}/hieradata`.

Now, we can create a simple `hiera.yaml` in `/etc/puppet/hiera.yaml` to show how the hierarchy is applied to a node, as shown here:

```
---
:hierarchy:
  - "hosts/%{::hostname}"
  - "roles/%{::role}"
  - "%{::kernel}/%{::os.family}/%{::os.release.major}"
  - "is_virtual/%{::is_virtual}"
  - common
:backends:
  - yaml
:yaml:
  :datadir:
```

This hierarchy is quite basic. Hiera will look for a variable starting with the hostname of the node in the host's directory and then move to the top scope variable role in the directory roles. If a value is not found in the roles, it will look in the `/etc/puppet/hieradata/kernel/osfamily/` directory (where `kernel` and `osfamily` will be replaced with the Facter values for these two facts) for a file named `lsbmajdistrelease.yaml`. On my test node, this would be `/etc/puppet/hieradata/Linux/RedHat/6.yaml`. If the value is not found there, then Hiera will continue to look in `hieradata/is_virtual/true.yaml` (as my node is a virtual machine, the value of `is_virtual` will be `true`). If the value is still not found, then the default file `common.yaml` will be tried. If the value is not found in `common`, then the command-line utility will return `nil`.

When using the `hiera` function in manifests, always set a default value, as failure to find anything in Hiera will lead to a failed catalog (although having the node fail when this happens is often used intentionally as an indicator of a problem).

As an example, we will set a variable `syslogpkg` to indicate which syslog package is used on our nodes. The syslog package is responsible for system logging. For EL7 and EL6 machines, the package is `rsyslog`; for EL5, the package is `syslog`. Create three YAML files, one for EL6 and EL7 at `/etc/puppetlabs/code/environments/production/hieradata/Linux/RedHat/7.yaml` using the following code:

```
---  
syslogpkg: rsyslog
```

Create another YAML file for EL5 at `/etc/puppetlabs/code/environments/production/hieradata/Linux/RedHat/5.yaml` using the following code:

```
---  
syslogpkg: syslog
```

With these files in place, we can test our Hiera by setting top scope variables (facts), using a YAML file. Create a `facts.yaml` file with the following content:

```
is_virtual: true  
kernel: Linux  
os:  
  family: RedHat  
  release:  
    major: "7"
```

We run Hiera three times, changing the value of `os.release.major` from 7 to 5 to 4 and observe the following results:

```
[thomas@stand ~]$ hiera syslogpkg -y facts.yaml environment=production  
rsyslog  
[thomas@stand ~]$ hiera syslogpkg -y facts.yaml environment=production  
syslog  
[thomas@stand ~]$ hiera syslogpkg -y facts.yaml environment=production  
nil
```

In the previous commands, we change the value of `os.release.major` from 7 to 5 to 4 to simulate the nodes running on EL7, EL5 and EL4. We do not have a `4.yaml` file, so there is no setting of `syslogpkg` and `hiera` that returns `nil`.

Now to use Hiera in our manifests, we can use the `hiera` function inline or set a variable using Hiera. When using Hiera, the syntax is `hiera('variable', 'default')`. The `variable` value is the key you are interested in looking at; the `default` value is the value to use when nothing is found in the hierarchy. Create a `syslog` module in `/etc/puppet/modules/syslog/manifest/init.pp` that starts `syslog` and makes sure the correct `syslog` is installed, as shown here:

```
class syslog {
  $syslogpkg = hiera('syslogpkg', 'syslog')
  package {"$syslogpkg":
    ensure => 'installed',
  }
  service {"$syslogpkg":
    ensure => true,
    enable => true,
  }
}
```

Then create an empty `/etc/puppet/manifests/site.pp` file that includes `syslog`, as shown here:

```
node default {
  include syslog
}
```

In this code, we set our `default` node to include the `syslog` module and then we define the `syslog` module. The `syslog` module looks for the Hiera variable `syslogpkg` to know which `syslog` package to install. Running this on our client node, we see that `rsyslog` is started as we are running EL7, as shown here:

```
[thomas@client ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.example.com
Info: Applying configuration version '1442381098'
Notice: /Stage[main]/Syslog/Package[rsyslog]/ensure: created
Notice: /Stage[main]/Syslog/Service[rsyslog]/ensure: ensure changed
'stopped' to 'running'
Info: /Stage[main]/Syslog/Service[rsyslog]: Unscheduling refresh on
Service[rsyslog]
Notice: Applied catalog in 7.83 seconds
```



If you haven't already disable the LDAP ENC, which we configured in the previous section, the instructions are provided at the end of the *LDAP backend* section of this chapter.

In the enterprise, you want a way to automatically apply classes to nodes based on facts. This is part of a larger issue of separating the code of your modules from the data used to apply them. We will examine this issue in greater depth in *Chapter 9, Roles and Profiles*. Hiera has a function that makes this very easy—`hiera_include`. Using `hiera_include`, you can have Hiera apply classes to a node based upon the hierarchy.

Using `hiera_include`

To use `hiera_include`, we set a Hiera variable to hold the name of the classes we would like to apply to the nodes. By convention, this is called `classes`, but it could be anything. We'll also set a variable `role` that we'll use in our new base class. We modify `site.pp` to include all the classes defined in the Hiera variable `classes`. We also set a default value if no values are found; this way we can guarantee that the catalogs will compile and all the nodes receive at least the base class. Edit `/etc/puppetlabs/code/environments/production/manifests/site.pp`, as follows:

```
node default {
  hiera_include('classes', 'base')
}
```

For the base class, we'll just set the `motd` file, as we've done previously. We'll also set a welcome string in Hiera. In `common.yaml`, we'll set this to something generic and override the value in a hostname-specific YAML file. Edit the base class in `/etc/puppetlabs/code/environments/production/modules/base/manifests/init.pp`, as follows:

```
class base {
  $welcome = hiera('welcome', 'Welcome')
  file {'/etc/motd':
    mode => '0644',
    owner => '0',
    group => '0',
    content => inline_template("<%= @welcome %>\nManaged Node: <%= @hostname %>\nManaged by Puppet version <%= @puppetversion %>\n"),
  }
}
```

This is our base class; it uses an inline template to set up the *message of the day* file (`/etc/motd`). We then need to set the welcome information in `hieradata`; edit `/etc/puppet/hieradata/common.yaml` to include the default welcome message, as shown here:

```
---
welcome: 'Welcome to Example.com'
classes:
  - 'base'
syslogpkg: 'nothing'
```

Now we can run Puppet on our `node1` machine. After the successful run, our `/etc/motd` file has the following content:

```
Welcome to Example.com
Managed Node: client
Managed by Puppet version 4.2.1
```

Now, to test if our hierarchy is working as expected, we'll create a YAML file specifically for `client`, `/etc/puppetlabs/code/environments/production/hieradata/hosts/client.yaml`, as follows:

```
---
welcome: 'Welcome to our default node'
```

Again, we run Puppet on `client` and examine the contents of `/etc/motd`, as shown here:

```
[thomas@client ~]$ cat /etc/motd
Welcome to our default node
Managed Node: client
Managed by Puppet version 4.2.1
```

Now that we have verified that our hierarchy performs as we expect, we can use Hiera to apply a class to all the nodes based on a fact. In this example, we'll use the `is_virtual` fact to do some performance tuning on our virtual machines. We'll create a virtual class in `/etc/puppet/modules/virtual/manifests/init.pp`, which installs the `tuned` package. It then sets the `tuned` profile to `virtual-guest` and starts the `tuned` service, as shown here:

```
class virtual {
  # performance tuning for virtual machine
  package {'tuned':
    ensure => 'present',
  }
```

```
service {'tuned':
  enable => true,
  ensure => true,
  require => Package['tuned']
}
exec {'set tuned profile':
  command => '/usr/sbin/tuned-adm profile virtual-guest',
  unless => '/bin/grep -q virtual-guest /etc/tune-profiles/
activeprofile',
}
}
```



In a real-world example, we'd verify that we only apply this to nodes running on EL6 or EL7.



This module ensures that the tuned package is installed and the tuned service is started. It then verifies that the current tuned profile is set to `virtual-guest` (using a `grep` statement in the `unless` parameter to the `exec`). If the current profile is not `virtual-guest`, the profile is changed to `virtual-guest` using `tuned-adm`.



Tuned is a tuning daemon included on enterprise Linux systems, which configures several kernel parameters related to scheduling and I/O operations.



To ensure that this class is applied to all virtual machines, we simply need to add it to the `classes` Hiera variable in `/etc/puppet/hieradata/is_virtual/true.yaml`, as shown here:

```
---
classes:
  - 'virtual'
```

Now our test node `client` is indeed virtual, so if we run Puppet now, the `virtual` class will be applied to the node and we will see that the tuned profile is set to `virtual-guest`. Running `tuned-adm active` on the host returns the currently active profile. When we run it initially, the command is not available as the `tuned` rpm has not been installed yet, as you can see here:

```
[thomas@client ~]$ sudo tuned-adm active
sudo: tuned-adm: command not found
```

Next, we run `puppet agent` to set the active profile (`tuned` is installed by default on EL7 systems):

```
[thomas@client ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.example.com
Info: Applying configuration version '1442383469'
Notice: /Stage[main]/Virtual/Exec[set tuned profile]/returns: executed
successfully
Notice: Applied catalog in 1.15 seconds
```

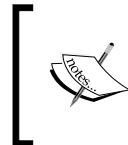
Now, when we run `tuned-adm active` we see that the active profile has been changed accordingly:

```
[thomas@client ~]$ sudo tuned-adm active
Current active profile: virtual-guest
```

This example shows the power of using Hiera, with `hiera_include` and a well-organized hierarchy. Using this method, we can have classes applied to nodes based on facts and reduce the need for custom classes on nodes. We do, however, have the option of adding classes per node since we have a `hosts/%{hostname}` entry in our hierarchy. If you had, for instance, a module that only needed to be installed on 32-bit systems, you could make an entry in `hiera.yaml` for `%{architecture}` and only create an `i686.yaml` file that contained the class in question. Building up your classes in this fashion reduces the complexity of your individual node configurations.

In fact, in Puppet version 3, architecture is available as both `architecture` and `os.architecture`.

Another great feature of Hiera is its ability to automatically fill in values for parameterized class attributes. For this example, we will create a class called `resolver` and set the search parameter for our `/etc/resolv.conf` file using **Augeas**.



Augeas is a tool to modify configuration files as though they were objects. For more information on Augeas, visit the project website at <http://augeas.net>. In this example, we will use Augeas to modify only a section of the `/etc/resolv.conf` file.

First, we will create a resolver class as follows in /etc/puppetlabs/code/environments/production/modules/resolver/manifests/init.pp:

```
class resolver($search = "example.com") {
  augeas { 'set resolv.conf search':
    context => '/files/etc/resolv.conf',
    changes => [
      "set search/domain '${search}'"
    ],
  }
}
```

Then we add resolver to our classes in /etc/puppetlabs/code/environments/production/hieradata/hosts/client.yaml, so as to have resolver applied to our node, as shown here:

```
---
welcome: 'Welcome to our default node'
classes:
- resolver
```

Now we run Puppet on the client; Augeas will change the resolv.conf file to have the search domain set to the default example.com:

```
[thomas@client ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.example.com
Info: Applying configuration version '1442411609'
Notice: Augeas[set resolv.conf search] (provider=augeas):
--- /etc/resolv.conf 2015-09-16 09:53:15.727804673 -0400
+++ /etc/resolv.conf.augnew 2015-09-16 09:53:28.108995714 -0400
@@ -2,3 +2,4 @@
nameserver 8.8.8.8
nameserver 8.8.4.4
domain example.com
+search example.com

Notice: /Stage[main]/Resolver/Augeas[set resolv.conf search]/returns:
executed successfully
Notice: Applied catalog in 1.41 seconds
```

Now, to get Hiera to override the default parameter for the parameterized class `resolver`, we simply set the Hiera variable `resolver::search` in our `/etc/puppetlabs/code/environments/production/hieradata/hosts/client.yaml` file, as shown here:

```
---  
welcome: 'Welcome to our default node'  
classes:  
  - resolver  
resolver::search: 'devel.example.com'
```

Running `puppet agent` another time on `node1` will change the search from `example.com` to `devel.example.com`, using the value from the Hiera hierarchy file, as you can see here:

```
[thomas@client ~]$ sudo puppet agent -t  
Info: Retrieving pluginfacts  
Info: Retrieving plugin  
Info: Caching catalog for client.example.com  
Info: Applying configuration version '1442411732'  
Notice: Augeas[set resolv.conf search] (provider=augeas):  
--- /etc/resolv.conf      2015-09-16 09:53:28.155509013 -0400  
+++ /etc/resolv.conf.augnew    2015-09-16 09:55:30.656393427 -0400  
@@ -2,4 +2,4 @@  
nameserver 8.8.8.8  
nameserver 8.8.4.4  
domain example.com  
-search example.com  
+search devel.example.com  
  
Notice: /Stage[main]/Resolver/Augeas[set resolv.conf search]/returns:  
executed successfully  
Notice: Applied catalog in 0.94 seconds
```

By building up your catalog in this fashion, it's possible to override parameters of any class. At this point, our `client` machine has the `virtual`, `resolver` and `base` classes, but our site manifest (`/etc/puppet/manifests/site.pp`) only has a `hiera_include` line, as shown here:

```
node default {  
  hiera_include('classes',base)  
}
```

In the enterprise, this means that you can add new hosts without modifying your site manifest and that you can customize the classes and any parameters to those classes.



Using `hiera_include` to specify the classes assigned to a node ensures that the node cannot assign itself to a different role. Some `site.pp` manifests will use a fact to determine the classes to be applied, this will allow anyone who can control facts on the node to change the classes on the node and potentially access configurations for different types of nodes.

Two other functions exist for using Hiera; they are `hiera_array` and `hiera_hash`. These functions do not stop at the first match found in Hiera and instead return either an array or hash of all the matches. This can also be used in powerful ways to build up definitions of variables. One good use of this is in setting the name servers a node will query. Using `hiera_array` instead of the `hiera` function, you can not only set nameservers based on the hostname of the node or some other facts, but also have the default nameservers from your `common.yaml` file applied to the node.

Summary

The classes that are applied to nodes should be as automatic as possible. Using a hostname convention and an ENC script, it is possible to have classes applied to nodes without any node-level configuration.

Using LDAP as a backend for class information may be a viable alternative at your enterprise. The LDAP schema included with Puppet can be successfully applied to an OpenLDAP instance or integrated into your existing LDAP infrastructure.

Hiera is a powerful tool to separate data from your module definitions. By utilizing a hierarchy of facts, it is possible to dynamically apply classes to nodes based on their facts.

The important concept in the enterprise is to minimize the customization required in the modules and push that customization up into the node declaration, to separate the code required to deploy your nodes from the specific data, through either LDAP, a custom ENC, or clever use of Hiera. If starting from scratch, Hiera is the most powerful and flexible solution to this problem.

In the next chapter, we will see how we can utilize Puppet environments to make Hiera even more flexible. We will cover using Git to keep our modules under version control.

3

Git and Environments

When working in a large organization, changes can break things. Every developer will need a sandbox to test their code. A single developer may have to work on two or three issues independently, throughout the day, but they may not apply the working code to any node. It would be great if you could work on a module and verify it in a development environment or even on a single node, before pushing it to the rest of your fleet. Environments allow you to carve up your fleet into as many development environments, as needed. Environments allow nodes to work from different versions of your code. Keeping track of the different versions with Git allows for some streamlined workflows. Other versioning systems can be used, but the bulk of integration in Puppet is done with Git.

Environments

When every node requests an object from the Puppet master, they inform the Puppet master of their environment. Depending on how the master is configured, the environment can change the set of modules, the contents of Hiera, or the site manifest (`site.pp`). The environment is set on the agent in their `puppet.conf` file or on the command line using the `puppet agent -environment` command.

In addition, environment may also be set from the ENC node terminus. In Puppet 4, setting the environment from the ENC overrides the setting in `puppet.conf`. If no environment is set, then production, which is the default environment, is applied.

In previous versions of Puppet, environments could be configured using section names in `puppet.conf` ([`production`] for example). In version 4 the only valid sections in `puppet.conf` are: `main`, `master`, `agent`, and `user`. Directory environments are now the only supported mechanism to configure environments. To configure directory environments, specify the `environmentpath` option in `puppet.conf`. The default `environmentpath` is `/etc/puppetlabs/code/environments`. The `production` environment is created by Puppet automatically. To create a development environment, create the `development` directory in `/etc/puppetlabs/code/environments`, as shown here:

```
[thomas@stand environments]$ sudo mkdir -p development/manifests  
development/modules
```

Now, to use the development environment, copy the base module from production to development, as shown here:

```
[thomas@stand environments]$ sudo cp -a production/modules/base  
development/modules
```

 In the remainder of this chapter, we will not use the ENC script we configured in *Chapter 2, Organizing Your Nodes and Data*. Modify `/etc/puppetlabs/puppet/puppet.conf` on `stand`, and comment out the two ENC-related settings which we configured in *Chapter 2, Organizing Your Nodes and Data*. My examples will be run on a standalone puppetserver machine named `stand`.

Next, modify the class definition in `/etc/puppetlabs/code/environments/development/modules/base/manifests/init.pp`, as follows:

```
class base {  
    $welcome = hiera('welcome', 'Unwelcome')  
    file {'/etc/motd':  
        mode    => '0644',  
        owner   => '0',  
        group   => '0',  
        content => inline_template("<%= @environment %>\n<%= @welcome  
%>\nManaged Node: <%= @hostname %>\nManaged by Puppet  
version <%= @puppetversion %>\n"),  
    }  
}
```

Now, run the `puppet agent` command on client and verify whether the production module is being used, as shown here:

```
[thomas@client ~]$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.example.com
Info: Applying configuration version '1442861899'
Notice: /Stage[main]/Base/File[/etc/motd]/ensure: defined content as '{md5}56289627b792b8ea3065b44c03b848a4'
Notice: Applied catalog in 0.84 seconds
[thomas@client ~]$ cat /etc/motd
Welcome to Example.com
Managed Node: client
Managed by Puppet version 4.2.1
```

Now, run the `puppet agent` command again with the `environment` option set to development, as shown here:

```
[thomas@stand ~]$ sudo puppet agent -t --environment=development
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for stand.example.com
Info: Applying configuration version '1442862701'
Notice: /Stage[main]/Base/File[/etc/motd]/content:
--- /etc/motd      2015-09-21 14:58:23.648809785 -0400
+++ /tmp/puppet-file20150921-3787-1aus09m          2015-09-21
15:11:41.753703780 -0400
@@ -1,3 +1,4 @@
-Welcome to Example.com
+development
+Unwelcome
Managed Node: stand
Managed by Puppet version 4.2.1

Info: Computing checksum on file /etc/motd
Info: /Stage[main]/Base/File[/etc/motd]: Filebucketed /etc/motd to puppet
with sum 56289627b792b8ea3065b44c03b848a4
```

```
Notice: /Stage[main]/Base/File[/etc/motd]/content: content
changed '{md5}56289627b792b8ea3065b44c03b848a4' to '{md5}
dd682631bcd240a08669c2c87a7e328d'
Notice: Applied catalog in 0.05 seconds
[thomas@stand ~]$ cat /etc/motd
development
Unwelcome
Managed Node: stand
Managed by Puppet version 4.2.1
```

This will perform a one-time compilation in the development environment. In the next Puppet run, when the environment is not explicitly set, this will default to production again. To permanently move the node to the development environment, edit `/etc/puppetlabs/puppet/puppet.conf` and set the environment, as shown here:

```
[agent]
environment = development
```

Environments and Hiera

Hiera's main configuration file can also use environment, as a variable. This leads us to two options: a single hierarchy with the environment as a hierarchy item and multiple hierarchies where the path to the `hieradata` directory comes from the environment setting. To have separate `hieradata` trees, you can use `environment` in the `datadir` setting for the backend, or to have parts of the hierarchy tied to your environment, put `%{::environment}` in the hierarchy.

Multiple hierarchies

To have a separate data tree, we will first copy the existing `hieradata` directory into the production and development directories, using the following commands:

```
Stand# cd /etc/puppetlabs/code/environments
stand# cp -a production/hieradata development
```

Now, edit `/etc/puppetlabs/puppet/hiera.yaml` and change `:datadir`, as follows:

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

Now, edit the welcome message in the `client.yaml` file of the production hieradata tree (`/etc/puppetlabs/code/environments/production/hieradata/hosts/client.yaml`), as shown here:

```
---
welcome: 'Production Node: watch your step.'
```

Also, edit the development hieradata tree (`/etc/puppetlabs/code/environments/development/hieradata/hosts/client.yaml`) to reflect the different environments, as shown here:

```
---
welcome: "Development Node: it can't rain all the time."
```

Now, run Puppet on `client` to see the `/etc/motd` file change according to the environment. First, we will run the agent without setting an environment so that the default setting of production is applied, as shown here:

```
[root@client ~]# puppet agent -t
...
Notice: /Stage[main]/Base/File[/etc/motd]/ensure: defined content as '{md
5}8147bf5dbb04eba29d5efb7e0fa28ce2'
Notice: Applied catalog in 0.83 seconds
[root@client ~]# cat /etc/motd
Production Node: watch your step.
Managed Node: client
Managed by Puppet version 4.2.2
```



If you have already set the environment value to development by adding `environment=development` in `puppet.conf`, remove that setting.

Then, we run the agent with environment set to development to see the change, as shown here:

```
[root@client ~]# puppet agent -t --environment=development
...
Notice: /Stage[main]/Base/File[/etc/motd]/content:
--- /etc/motd    2015-09-30 21:35:59.523156901 -0700
+++ /tmp/puppet-file20150930-3185-1vyeusl      2015-09-30
21:45:18.444186406 -0700
@@ -1,3 +1,3 @@
@@
```

```
-Production Node: watch your step.  
+Development Node: it can't rain all the time.  
Managed Node: client  
Managed by Puppet version 4.2.2  
  
Info: Computing checksum on file /etc/motd  
Info: /Stage[main]/Base/File[/etc/motd]: Filebucketed /etc/motd to puppet  
with sum 8147bf5dbb04eba29d5efb7e0fa28ce2  
Notice: /Stage[main]/Base/File[/etc/motd]/content: content  
changed '{md5}8147bf5dbb04eba29d5efb7e0fa28ce2' to '{md5}  
dc504aeaeb934ad078db900a97360964'  
Notice: Applied catalog in 0.04 seconds
```

Configuring Hiera in this fashion will let you keep completely distinct hieradata trees, for each environment. You can, however, configure Hiera to look for environment-specific information in a single tree.

Single hierarchy for all environments

To have one hierarchy for all environments, edit `hiera.yaml` as follows:

```
---  
:hierarchy:  
  - "environments/%{environment}"  
  - "hosts/%{hostname}"  
  - "roles/%{role}"  
  - "%{kernel}/%{os.family}/%{os.release.major}"  
  - "is_virtual/%{is_virtual}"  
  - common  
:backends:  
  - yaml  
:yaml:  
  :datadir: '/etc/puppetlabs/code/hieradata'
```

Next, create an `environments` directory in `/etc/puppetlabs/code/hieradata` and create the following two YAML files: one for production (`/etc/puppetlabs/code/hieradata/environments/production.yaml`) and another for development (`/etc/puppetlabs/code/hieradata/environments/development.yaml`). The following will be the welcome message for the production file:

```
---  
welcome: 'Single tree production welcome'
```

The following will be the welcome message for the development file:

```
---  
welcome: 'Development in Single Tree'
```

Restart httpd on stand and run Puppet on node1 again to see the new motd for production, as shown here:

```
[root@client ~]# puppet agent -t  
...  
Notice: /Stage[main]/Base/File[/etc/motd]/content:  
--- /etc/motd    2015-09-30 21:45:18.465186407 -0700  
+++ /tmp/puppet-file20150930-3221-bh9ik5          2015-09-30  
21:53:43.283213056 -0700  
@@ -1,3 +1,3 @@  
-Development Node: it can't rain all the time.  
+Single tree production welcome  
...  
Notice: Applied catalog in 0.67 seconds
```

Having the production and development environments may be sufficient for a small operation (a manageable amount of nodes, typically less than a thousand), but in an enterprise, you will need many more such environments to help admins avoid stumbling upon one another.

Previous versions of Puppet required special configuration to work with arbitrary environment names; this is no longer the case. The current state of environments is directory environments; any subdirectory within the environmentpath configuration variable is a valid environment. To create new environments, you need to only create the directory within the environmentpath directory. By default, the environmentpath variable is set to /etc/puppetlabs/code/environments. The code directory is meant to be a catchall location for your code. This change was made to help us separate code from configuration data.

Directory environments

Our configuration for Hiera did not specify production or development environments in `hiera.yaml`. We used the `environment` value to fill in a path on the filesystem. The directory environments in Puppet function the same way. Directory environments are the preferred method for configuring environments. When using directory environments, it is important to always account for the production environment, since it is the default setting for any node, when environment is not explicitly set.

Puppet determines where to look for directory environments, based on the `environmentpath` configuration variable. In Puppet 4, the default `environmentpath` is `/etc/puppetlabs/code`. Within each environment, an `environment.conf` file can be used to specify `modulepath`, `manifest`, `config_version` and `environment_timeout` per environment. When Puppet compiles a catalog, it will run the script specified by `config_version` and use the output as the config version of the catalog. This provides a mechanism to determine which code was used to compile a specific catalog.

Versions 3.7 and above of Puppet also support a parser setting, which determines which parser (current or future) is used to compile the catalog in each environment. In version 4 and above, the future parser is the only available parser. Using the parser setting, it is possible to test your code against the future parser before upgrading from Puppet 3.x to 4. The usage scenario for the `parser` option will be to upgrade your development environments to the future parser and fix any problems you encounter along the way. You will then promote your code up to production and enable the future parser in production.

Using the `manifest` option in `environment.conf`, it is possible to have a per environment site manifests. If your enterprise requires a site manifest to be consistent between environments, you can set `disable_per_environment_manifest = true` in `puppet.conf` to use the same site manifest for all environments.

If an environment does not specify a manifest in `environment.conf`, the manifest specified in `default_manifest` is used. A good use of this setting is to specify a default manifest and not specify one within your `environment.conf` files. You can then test a new site manifest in an environment by adding it to the `environment.conf` within that environment.

The `modulepath` within `environment.conf` may have relative paths and absolute paths. To illustrate, consider the environment named `mastering`. In the `mastering` directory within `environmentpath` (`/etc/puppetlabs/code/environments`), the `environment.conf` file has `modulepath` set to `modules:public:/etc/puppetlabs/code/modules`. When searching for modules in the `mastering` environment, Puppet will search the following locations:

- `/etc/puppetlabs/code/environments/mastering/modules`
- `/etc/puppetlabs/code/environments/mastering/public`
- `/etc/puppetlabs/code/modules`

Another useful setting in `puppet.conf` is `basemodulepath`. You may configure `basemodulepath` to a set of directories containing your enterprise wide modules or modules that are shared across multiple environments. The `$basemodulepath` variable is available within `environment.conf`. A typical usage scenario is to have `modulepath` within `environment.conf` configured with `$basemodulepath`, defined first in the list of directories, as shown here:

```
modulepath=$basemodulepath:modules:site:/etc/puppetlabs/code/modules
```

A great use of directory environments is to create test environments where you can experiment with modifications to your site manifest without affecting other environments. (provided you haven't used the `disable_per_environment_manifest` setting). As an example, we'll create a sandbox environment and modify the manifest within that environment.

Create the `sandbox` directory and `environment.conf` with the following contents:

```
manifest=/etc/puppetlabs/code/test
```

Now, create the site manifest at `/etc/puppetlabs/code/test/site.pp` with the following code:

```
node default {
  notify { "Trouble in the Henhouse": }
}
```

Now when you do an agent run on the client node against the sandbox environment, the site manifest in `/etc/puppetlabs/code/test` will be used, as shown here:

```
[root@client ~]# puppet agent -t --environment sandbox
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.local
Info: Applying configuration version '1443285153'
Notice: Trouble in the Henhouse
Notice: /Stage[main]/Main/Node[default]/Notify[Trouble in the Henhouse]/message: defined 'message' as 'Trouble in the Henhouse'
Notice: Applied catalog in 0.02 seconds
```

This type of playing around with environments is great for a single developer, but when you are working in a large team, you'll need some version control and automation to convert this to a workable solution. With a large team, it is important that admins do not interfere with each other when making changes to the `/etc/puppetlabs/code` directory. In the next section, we'll use Git to automatically create environments and share environments between developers.

For further reading on environments, refer to the Puppet Labs website at <http://docs.puppetlabs.com/guides/environment.html>.

Git

Git is a version control system written by Linus Torvalds, which is used to collaborate development on the Linux kernel source code. Its support for rapid branching and merging makes it the perfect choice for a Puppet implementation. In Git, each source code commit has references to its parent commit; to reconstruct a branch, you only need to follow the commit trail back. We will be exploiting the rapid branch support to have environments defined from Git branches.



It is possible to use Git without a server and to make copies of repositories using only local Git commands.



In your organization, you are likely to have some version control software. The software in question isn't too important, but the methodology used is important.



Remember that passwords and sensitive information stored in version control will be available to anyone with access to your repository. Also, once stored in version control, it will always be available.



Long running branches or a stable trunk are the terms used in the industry to describe the development cycle. In our implementation, we will assume that development and production are long running branches. By long running, we mean that these branches will persist throughout the lifetime of the repository. Most of the other branches are dead ends – they solve an immediate issue and then get merged into the long running branches and cease to exist, or they fail to solve the issue and are destroyed.

Why Git?

Git is the de facto version control software with Puppet because of its implementation of rapid branching. There are numerous other reasons for using Git in general. Each user of Git is given a complete copy of the revision history whenever they clone a Git repository. Each developer is capable of acting as a backup for the repository, should the need arise. Git allows each developer to work independently from the master repository; thus, allowing developers to work offsite and even without network connectivity.

This section isn't intended to be an exhaustive guide of using Git. We'll cover enough commands to get your job done, but I recommend that you do some reading on the subject to get well acquainted with the tool.



The main page for Git documentation is <http://git-scm.com/documentation>. Also worth reading is the information on getting started with Git by GitHub at <http://try.github.io> or the *Git for Ages 4 and Up* video available at http://mirror.int.linux.conf.au/linux.conf.au/2013/ogg/Git_For_Ages_4_And_Up.ogg.

To get started with Git, we need to create a bare repository. By bare, we mean that only the meta information and checksums will be stored; the files will be in the repository but only in the checksum form. Only the main location for the repository needs to be stored in this fashion.

In the enterprise, you want the Git server to be a separate machine, independent of your Puppet master. Perhaps, your Git server isn't even specific for your Puppet implementation. The great thing about Git is that it doesn't really matter at this point; we can put the repository wherever we wish.

To make things easier to understand, we'll work on our single worker machine for now, and in the final section, we will create a new Git server to hold our Git repository.



GitHub or GitHub Enterprise can also be used to host Git repositories. GitHub is a public service but it also has pay account services. GitHub Enterprise is an appliance solution to host the same services as GitHub internally, within your organization.

A simple Git workflow

On our standalone machine, install Git using yum, as shown here:

```
[root@stand ~]# yum install -y git  
...  
Installed: git.x86_64 0:1.8.3.1-5.el7
```

Now, decide on a directory to hold all your Git repositories. We'll use `/var/lib/git` in this example.

 A directory under `/srv` may be more appropriate for your organization. Several organizations have adopted the `/apps` directory for application specific data, as well and using these locations may have SELinux context considerations. The targeted policy on RedHat systems provides for the `/var/lib/git` and `/var/www/git` locations for Git repository data.

The `/var/lib/git` path closely resembles the paths used by other EL packages. Since running everything as root is unnecessary, we will create a Git user and make that user the owner of the Git repositories.

Create the directory to contain our repository first (`/var/lib/git`) and then create an empty Git repository (using the `git init --bare` command) in that location, as shown in the following code:

```
[root@stand ~]# useradd git -c 'Git Repository Owner' -d /var/lib/git  
[root@stand ~]# sudo -iu git  
[git@stand ~]$ pwd  
/var/lib/git  
[git@stand ~]$ chmod 755 /var/lib/git  
[git@stand ~]$ git init --bare control.git  
Initialized empty Git repository in /var/lib/git/control.git/  
[git@stand ~]$ cd /tmp  
[git@stand tmp]$ git clone /var/lib/git/control.git  
Cloning into 'control'...  
warning: You appear to have cloned an empty repository.  
done.  
[git@stand tmp]$ cd control  
[git@stand control]$ git status
```

```
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```



Using `git --bare` will create a special copy of the repository where the code is not checked out; it is known as bare because it is without a working copy of the code. A normal copy of a Git repository will have the code available at the top-level directory and the Git internal files in a `.git` directory. A bare repository has the contents of the `.git` directory located in the top level directory.

Now that our repository is created, we should start adding files to the repository. However, we should first configure Git. Git will store our username and e-mail address with each commit. These settings are controlled with the `git config` command. We will add the `--global` option to ensure that the config file in `~/.git` is modified, as shown in the following example:

```
[git@stand ~]$ git config --global user.name 'Git Repository Owner'
[git@stand ~]$ git config --global user.email 'git@example.com'
```

Now, we'll copy in our production modules and commit them. We'll copy the files from the `/etc/puppet/environments/production` directory of our worker machines and then add them to the repository using the `git add` command, as shown here:

```
[git@stand ~]$ cd /tmp/control/
[git@stand control]$ cp -a /etc/puppetlabs/code/environments/production/*
.
[git@stand control]$ ls environment.conf hieradata manifests modules
[git@stand control]$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       environment.conf
```

```
#      hieradata/
#      manifests/

#      modules/
nothing added to commit but untracked files present (use "git add" to
track)
```

We've copied our `hieradata`, `manifests`, and `modules` directories, but Git doesn't know anything about them. We now need to add them to the Git repository and commit to the default branch `master`. This is done with two Git commands, first using `git add` and then using `git commit`, as shown in the following code:

```
[git@stand control]$ git add hieradata manifests modules environment.conf
[git@stand control]$ git commit -m "initial commit"
[master (root-commit) 316a391] initial commit
 14 files changed, 98 insertions(+)
...
create mode 100644 modules/web/manifests/init.pp
```

 To see the files that will be committed when you issue `git commit`, use `git status` after the `git add` command. It is possible to commit in a single command using `git commit -a`. This will commit all staged files (these are the files that have changed since the last commit). I prefer to execute the commands separately to specifically add the files, which I would like to add in the commit. If you are editing a file with vim, you may inadvertently commit a swap file using `git commit -a`.

At this point, we've committed our changes to our local copy of the repository. To ensure that we understand what is happening, we'll clone the initial location again into another directory (`/tmp/control2`), using the following commands:

```
[git@stand control]$ cd /tmp
[git@stand tmp]$ mkdir control2
[git@stand tmp]$ git clone /var/lib/git/control.git .
fatal: destination path '.' already exists and is not an empty directory.
[git@stand tmp]$ cd control2
[git@stand control2]$ git clone /var/lib/git/control.git .
Cloning into '...'...
warning: You appear to have cloned an empty repository.
done.
[git@stand control2]$ ls
```

Our second copy doesn't have the files we just committed, and they only exist in the first local copy of the repository. One of the most powerful features of Git is that it is a self-contained environment. Going back to our first clone (`/tmp/control`), examine the contents of the `.git/config` file. The `url` setting for `remote "origin"` points to the remote master that our repository is based on (`/var/lib/git/control.git`), as shown in the following code:

```
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
[remote "origin"]
url = /var/lib/git/control.git
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
remote = origin
merge = refs/heads/master
```

In Git, `origin` is where the original remote repository lives. In this example, it is a local location (`/var/lib/git/control.git`), but it can also be an HTTPS URI or SSH URI.

To push the local changes to the remote repository, we use `git push`; the default push operation is to push it to the first `[remote]` repository (named `origin` by default) to the currently selected branch (the current branch is given in the output from `git status`). The default branch in Git is named `master`, as we can see in the `[branch "master"]` section. To emphasize what we are doing, we'll type in the full arguments to push (although `git push` will achieve the same result in this case), as you can see here:

```
[git@stand control]$ cd
[git@stand ~]$ cd /tmp/control
[git@stand control]$ git push origin master
Counting objects: 34, done.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (34/34), 3.11 KiB | 0 bytes/s, done.
Total 34 (delta 0), reused 0 (delta 0)
To /var/lib/git/control.git
 * [new branch]      master -> master
```

Now, even though our remote repository has the updates, they are still not available in our second copy (`/tmp/control2`). We must now pull the changes from the origin to our second copy using `git pull`. Again, we will type in the full argument list (this time, `git pull` will do the same thing), as shown here:

```
[git@stand ~]$ cd /tmp/control
[git@stand control]$ git push origin master
Counting objects: 34, done.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (34/34), 3.11 KiB | 0 bytes/s, done.
Total 34 (delta 0), reused 0 (delta 0)
To /var/lib/git/control.git
 * [new branch]      master -> master
[git@stand control]$ cd
[git@stand ~]$ cd /tmp/control2
[git@stand control2]$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
[git@stand control2]$ ls
[git@stand control2]$ git pull origin master
remote: Counting objects: 34, done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 34 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (34/34), done.
From /var/lib/git/control
 * branch      master      -> FETCH_HEAD
[git@stand control2]$ ls
environment.conf  hieradata  manifests  modules
```

Two useful commands that you should know at this point are `git log` and `git show`. The `git log` command will show you the log entries from Git commits. Using the log entries, you can run `git show` to piece together what your fellow developers have been doing. The following snippet shows the use of the following two commands in our example:

```
[git@stand control2]$ git log
commit 316a391e2641dd9e44d2b366769a64e88cc9c557
Author: Git Repository Owner <git@example.com>
```

```
Date: Sat Sep 26 19:13:41 2015 -0400

    initial commit

[git@stand control]$ git show 316a391e2641dd9e44d2b366769a64e88cc9c557
commit 316a391e2641dd9e44d2b366769a64e88cc9c557
Author: Git Repository Owner <git@example.com>
Date:   Sat Sep 26 19:13:41 2015 -0400

    initial commit

diff --git a/environment.conf b/environment.conf
new file mode 100644
index 0000000..c39193f
--- /dev/null
+++ b/environment.conf
@@ -0,0 +1,18 @@
+# Each environment can have an environment.conf file. Its settings will
only
+# affect its own environment. See docs for more info:
+# https://docs.puppetlabs.com/puppet/latest/reference/config_file_
environment.html
...

```

The `git show` command takes the commit hash as an optional argument and returns all the changes that were made with that hash.

Now that we have our code in the repository, we need to create a production branch for our production code. Branches are created using `git branch`. The important concept to be noted is that they are local until they are pushed to the origin. When `git branch` is run without arguments, it returns the list of available branches with the currently selected branch highlighted with an asterisk, as shown here:

```
[git@stand ~]$ cd /tmp/control
[git@stand control]$ git branch
* master
[git@stand control]$ git branch production
[git@stand control]$ git branch
* master
production
```

This sometimes confuses people. You have to checkout the newly created branch after creating it. You can do this in one step using the `git checkout -b <branch_name>` command, but I believe using this shorthand command initially leads to confusion. We'll now check our production branch and make a change. We will then commit to the local repository and push to the remote, as shown here:

```
[git@stand control]$ git checkout production
Switched to branch 'production'
[git@stand control]$ git branch
  master
* production
[git@stand control]$ cd hieradata/hosts/
[git@stand hosts]$ sed -i -e 's/watch your step/best behaviour/' client.yaml
[git@stand hosts]$ git add client.yaml
[git@stand hosts]$ git commit -m "modifying welcome message on client"
[production 74d2ff5] modifying welcome message on client
  1 file changed, 1 insertion(+), 1 deletion(-)
[git@stand hosts]$ git push origin production
Counting objects: 9, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 569 bytes | 0 bytes/s, done.
Total 5 (delta 1), reused 0 (delta 0)
To /var/lib/git/control.git
 * [new branch]      production -> production n
```

Now, in our second copy of the repository, let's confirm that the production branch has been added to the origin, using `git fetch` to retrieve the latest metadata from the remote origin, as shown here:

```
[git@stand hosts]$ cd /tmp/control2/
[git@stand control2]$ git branch -a
* master
[git@stand control2]$ git fetch
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From /var/lib/git/control
```

```
* [new branch]      master      -> origin/master
* [new branch]      production -> origin/production
[git@stand control2]$ git branch -a
* master
  remotes/origin/master
  remotes/origin/production
```

It is important to run `git fetch` routinely, to take a look at the changes that your teammates may have made and branches that they may have created. Now, we can verify whether the production branch has the change we made. The `-a` option to `git branch` instructs Git to include remote branches in the output. We'll display the current contents of `client.yaml` and then run `git checkout production` to see the production version, as shown here:

```
[git@stand control2]$ grep welcome hieradata/hosts/client.yaml
welcome: 'Production Node: watch your step.'
[git@stand control2]$ git checkout production
Branch production set up to track remote branch production from origin.
Switched to a new branch 'production'
[git@stand control2]$ grep welcome hieradata/hosts/client.yaml
welcome: 'Production Node: best behaviour.'
```

As we can see, the welcome message in the production branch is different from that of the master branch. At this point, we'd like to have the production branch in `/etc/puppetlabs/code/environments/production` and the master branch in `/etc/puppetlabs/code/environments/master`. We'll perform these commands, as the root user, for now:

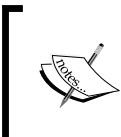
```
[root@stand ~]# cd /etc/puppetlabs/code/
[root@stand code]# mv environments environments.orig
[root@stand code]# mkdir environments
[root@stand code]# cd environments
[root@stand environments]# for branch in production master
> do
> git clone -b $branch /var/lib/git/control.git $branch
> done
Cloning into 'production'...
done.
Cloning into 'master'...
done.
```

Now that our production branch is synchronized with the remote, we can do the same for the master branch and verify whether the branches differ, using the following command:

```
[root@stand environments]# diff production/hieradata/hosts/client.yaml  
master/hieradata/hosts/client.yaml  
  
2c2  
< welcome: 'Production Node: best behaviour.'  
---  
> welcome: 'Production Node: watch your step.'
```

Running Puppet on client in the production environment will now produce the change we expect in /etc/motd, as follows:

```
Production Node: best behaviour.  
Managed Node: client  
Managed by Puppet version 4.2.2
```



If you changed hiera.yaml for the single tree example, change it to the following:

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



Run the agent again with the master environment, to change motd, as shown here:

```
[root@client ~]# puppet agent -t --environment master  
Info: Retrieving pluginfacts  
Info: Retrieving plugin  
Info: Caching catalog for client.example.com  
Info: Applying configuration version '1443313038'  
Notice: /Stage[main]/Virtual/Exec[set tuned profile]/returns: executed  
successfully  
Notice: /Stage[main]/Base/File[/etc/motd]/content:  
--- /etc/motd    2015-10-01 22:23:02.786866895 -0700  
+++ /tmp/puppet-file20151001-12407-16iuoej      2015-10-01  
22:24:02.999870073 -0700  
@@ -1,3 +1,3 @@  
-Production Node: best behaviour.  
+Production Node: watch your step.  
Managed Node: client  
Managed by Puppet version 4.2.2
```

```
Info: Computing checksum on file /etc/motd
Info: /Stage[main]/Base/File[/etc/motd]: Filebucketed /etc/motd to puppet
with sum 490af0a672e3c3fdc9a3b6e1bf1f1c7b
Notice: /Stage[main]/Base/File[/etc/motd]/content: content changed '{md5}
490af0a672e3c3fdc9a3b6e1bf1f1c7b' to '{md5}8147bf5dbb04eba29d5efb7e0fa28
ce2'
Notice: Applied catalog in 1.07 seconds
```

Our standalone Puppet master is now configured such that each branch of our control repository is mapped to a separate Puppet environment. As new branches are added, we have to set up the directory manually and push the contents to the new directory. If we were working in a small environment, this arrangement of Git pulls will be fine; but, in an enterprise, we would want this to be automatic. In a large environment, you would also want to define your branching model to ensure that all your team members are working with branches in the same way. Good places to look for branching models are <http://nvie.com/posts/a-successful-git-branching-model/> and <https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows>. Git can run scripts at various points in the commitment of code to the repository – these scripts are called hooks.

Git hooks

Git provides several hook locations that are documented in the `githooks` man page. The hooks of interest are `post-receive` and `pre-receive`. A `post-receive` hook is run after a successful commit to the repository and a `pre-receive` hook is run before any commit is attempted. Git hooks can be written in any language; the only requirement is that they should be executable.

Each time you commit to a Git repository, Git will create a hash that is used to reference the state of the repository after the commit. These hashes are used as references to the state of the repository. A branch in Git refers to a specific hash, you can view this hash by looking at the contents of `.git/HEAD`, as shown here:

```
[root@stand production]# cat .git/HEAD
ref: refs/heads/production
```

The hash will be in the file located at `.git/refs/heads/production`, as shown here:

```
[root@stand production]# cat .git/refs/heads/production
74d2ff58470d009e96d9ea11b9c126099c9e435a
```

The post-receive and pre-receive hooks are both passed three parameters via `stdin`: the first is the commit hash that you are starting from (`oldrev`), the second is the new commit hash that you are creating (`newrev`), and the third is a reference to the type of change that was made to the repository, where the reference is the branch that was updated. Using these hooks, we can automate our workflow. We'll start using the post-receive hook to set up our environments for us.

Using post-receive to set up environments

What we would like to happen at this point is a series of steps discussed as follows:

1. A developer works on a file in a branch.
2. The developer commits the change and pushes it to the origin.
3. If the branch doesn't exist, create it in `/etc/puppetlabs/code/environments/<branch>`.
4. Pull the updates for the branch into `/etc/puppetlabs/code/environments/<branch>`.

In our initial configuration, we will write a post-receive hook that will implement steps 3 and 4 mentioned previously. Later, we'll ensure that only the correct developers commit to the correct branch with a pre-receive hook. To ensure that our Puppet user has access to the files in `/etc/puppetlabs/code/environments`, we will use the `sudo` utility to run the commits, as the Puppet user.

Our hook doesn't need to do anything with the reference other than extract the name of the branch and then update `/etc/puppetlabs/code/environments`, as necessary. To maintain the simplicity, this hook will be written in bash. Create the script in `/var/lib/git/control.git/hooks/post-receive`, as follows:

```
#!/bin/bash
PUPPETDIR=/etc/puppetlabs/code/environments
REPOHOME=/var/lib/git/control.git
GIT=/bin/git
umask 0002
unset GIT_DIR
```

We will start by setting some variables for the Git repository location and Puppet environments directory location. It will become clear later why we set `umask` at this point, we want the files created by our script to be group writable. The `unset GIT_DIR` line is important; the hook will be run by Git after a successful commit, and during the commit `GIT_DIR` is set to `"."`. We unset the variable so that Git doesn't get confused.

Next, we will read the variables `oldrev`, `newrev`, and `refname` from `stdin` (not command-line arguments), as shown in the following code:

```
read oldrev newrev refname
branch=${refname##*//**/}
if [ -z $branch ]; then
    echo "ERROR: Updating $PUPPETDIR"
    echo "      Branch undefined"
    exit 10
fi
```

After extracting the branch from the third argument, we will verify whether we were able to extract a branch. If we are unable to parse out the branch name, we will quit the script and warn the user.

Now, we have three scenarios that we will account for in the script. The first is that the directory exists in `/etc/puppetlabs/code/environments` and that it is a Git repository, as shown:

```
# if directory exists, check it is a git repository
if [ -d "$PUPPETDIR/$branch/.git" ]; then
    cd $PUPPETDIR/$branch
    echo "Updating $branch in $PUPPETDIR"
    sudo -u puppet $GIT pull origin $branch
    exit=$?
```

In this case, we will `cd` to the directory and issue a `git pull origin <branchname>` command to update the directory. We will run the `git pull` command using `sudo` with `-u puppet` to ensure that the files are created as the Puppet user.

The second scenario is that the directory exists but it was not created via a Git checkout. We will quit early if we run into this option, as shown in the following snippet:

```
elif [ -d "$PUPPETDIR/$branch" ]; then
    # directory exists but is not in git
    echo "ERROR: Updating $PUPPETDIR"
    echo "      $PUPPETDIR/$branch is not a git repository"
    exit=20
```

The third option is that the directory doesn't exist yet. In this case, we will clone the branch using the `git clone` command in a new directory, as the Puppet user (using `sudo` again), as shown in the following snippet:

```
else
    # directory does not exist, create
    cd $PUPPETDIR
```

```
echo "Creating new branch $branch in $PUPPETDIR"
sudo -u puppet $GIT clone -b $branch $REPOHOME $branch
exit=$?
fi
```

In each case, we retained the return value from Git so that we can exit the script with the appropriate exit code at this point, as follows:

```
exit $exit
```

Now, let's see this in action. Change the permissions on the post-receive script to make it executable (`chmod 755 post-receive`). Now, to ensure that our Git user can run the Git commands as the Puppet user, we need to create a sudoers file. We need the Git user to run `/usr/bin/git`; so, we put in a rule to allow this in a new file called `/etc/sudoers.d/sudoers-puppet`, as follows:

```
git ALL = (puppet) NOPASSWD: /bin/git *
```

In this example, we'll create a new local branch, make a change in the branch, and then push the change to the origin. Our hook will be called and a new directory will be created in `/etc/puppet/environments`.

```
[root@stand ~]# chown puppet /etc/puppetlabs/code/environments
[root@stand ~]# sudo -iu git
[git@stand ~]$ ls /etc/puppetlabs/code/environments
master production
[git@stand ~]$ cd /tmp/control
[git@stand control]$ git branch thomas
[git@stand control]$ git checkout thomas
Switched to branch 'thomas'
1 files changed, 1 insertions(+), 1 deletions(-)
[git@stand control]$ sed -i hieradata/hosts/client.yaml -e "s/welcome:.*/
welcome: 'Thomas Branch'/"
[git@stand control]$ git add hieradata/hosts/client.yaml
[git@stand control]$ git commit -m "Creating thomas branch"
[thomas 598d13b] Creating Thomas branch
1 file changed, 1 insertion(+)
[git@stand control]$ git push origin thomas
Counting objects: 9, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 501 bytes | 0 bytes/s, done.
Total 5 (delta 2), reused 0 (delta 0)
```

```
To /var/lib/git/control.git
 * [new branch]      thomas -> thomas
remote: Creating new branch thomas in /etc/puppetlabs/code/environments
remote: Cloning into 'thomas'...
remote: done.

To /var/lib/git/control.git
    b0fc881..598d13b  thomas -> thomas
[git@stand control]$ ls /etc/puppetlabs/code/environments
master  production  thomas
```

Our Git hook has now created a new environment, without our intervention. We'll now run `puppet agent` on the node to see the new environment in action, as shown here:

```
[root@client ~]# puppet agent -t --environment thomas
...
Notice: /Stage[main]/Base/File[/etc/motd]/content:
--- /etc/motd    2015-10-01 22:24:03.057870076 -0700
+++ /tmp/puppet-file20151001-12501-1y5t102      2015-10-01
22:55:59.132971224 -0700
@@ -1,3 +1,3 @@
-Production Node: watch your step.
+Thomas Branch
...
Notice: Applied catalog in 1.78 seconds
```

Our post-receive hook is very simple, but it illustrates the power of automating your code updates. When working in an enterprise, it's important to automate all the processes that take your code from development to production. In the next section, we'll look at a community solution to the Git hook problem.

Puppet-sync

The problem of synchronizing Git repositories for Puppet is common enough that a script exists on GitHub that can be used for this purpose. The `puppet-sync` script is available at <https://github.com/pdxcat/puppet-sync>.

To quickly install the script, download the file from GitHub using curl, and redirect the output to a file, as shown here:

```
[root@stand ~]# curl https://raw.githubusercontent.com/pdxcat/puppet-sync/4201dbe7af4ca354363975563e056edf89728dd0/puppet-sync >/usr/bin/puppet-sync
          % Total     % Received % Xferd  Average Speed   Time     Time     Time
Current                                            Dload  Upload   Total   Spent   Left
Speed
100  7246  100  7246    0      0  9382       0  --::--  --::--  --::--
-  9373
[root@stand ~]# chmod 755 /usr/bin/puppet-sync
```

To use puppet-sync, you need to install the script on your master machine and edit the post-receive hook to run puppet-sync with appropriate arguments. The updated post-receive hook will have the following lines:

```
#!/bin/bash
PUPPETDIR=/etc/puppetlabs/code/environments
REPOHOME=/var/lib/git/control.git

read oldrev newrev refname
branch=${refname##*//*}
if [ -z "$branch" ]; then
    echo "ERROR: Updating $PUPPETDIR"
    echo "        Branch undefined"
    exit 10
fi

[ "$newrev" -eq 0 ] 2> /dev/null && DELETE='--delete' || DELETE=''
sudo -u puppet /usr/bin/puppet-sync \
    --branch "$branch" \
    --repository "$REPOHOME" \
    --deploy "$PUPPETDIR" \
    $DELETE
```

To use this script, we will need to modify our sudoers file to allow Git to run puppet-sync as the Puppet user, as shown:

```
git ALL = (puppet) NOPASSWD: /bin/git *, /usr/bin/puppet-sync *
```

This process can be extended, as a solution, to push across multiple Puppet masters by placing a call to puppet-sync within a for loop, which SSHes to each worker and then runs puppet-sync on each of them.

This can be extended further by replacing the call to `puppet-sync` with a call to Ansible, to update a group of Puppet workers defined in your Ansible host's file. More information on Ansible is available at <http://docs.ansible.com/>.

To check whether `puppet-sync` is working as expected, create another branch and push it back to the origin, as shown:

```
[root@stand hooks]# sudo -iu git
[git@stand ~]$ cd /tmp/control
[git@stand control]$ git branch puppet_sync
[git@stand control]$ git checkout puppet_sync
Switched to branch 'puppet_sync'
[git@stand control]$ sed -i hieradata/hosts/client.yaml -e "s/welcome:.*/
welcome: 'Puppet_Sync Branch, underscores are cool.'/"
[git@stand control]$ git add hieradata/hosts/client.yaml
[git@stand control]$ git commit -m "creating puppet_sync branch"
[puppet_sync e3dd4a8] creating puppet_sync branch
 1 file changed, 1 insertion(+), 1 deletion(-)
[git@stand control]$ git push origin puppet_sync
Counting objects: 9, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 499 bytes | 0 bytes/s, done.
Total 5 (delta 2), reused 0 (delta 0)
remote: ----- PuppetSync -----
remote: | Host      : stand.example.com
remote: | Branch    : puppet_sync
remote: | Deploy To : /etc/puppetlabs/code/environments/puppet_sync
remote: | Repository : /var/lib/git/control.git
remote: `-----
To /var/lib/git/control.git
e96c344..6efa315  puppet_sync -> puppet_sync
```

In a production environment, this level of detail for every commit will become cumbersome, `puppet-sync` has a quiet option for this purpose; add `-q` to your post-receive call to `puppet-sync` to enable the quiet mode.

Puppet environments must start with an alphabetic character and only contain alphabetic characters, numbers, and the underscore. If we name our branch `puppet-sync`, it will produce an error when attempting to execute `puppet agent -t -environment puppet-sync`.

Using Git hooks to play nice with other developers

Up to this point, we've been working with the Git account to make our changes. In the real world, we want the developers to work with their own user account. We need to worry about permissions at this point. When each developer commits their code, the commit will run as their user; so, the files will get created with them as the owner, which might prevent other developers from pushing additional updates. Our post-receive hook will run as their user, so they need to be able to use sudo just like the Git user. To mitigate some of these issues, we'll use Git's sharedrepository setting to ensure that the files are group readable in `/var/lib/git/control.git`, and use sudo to ensure that the files in `/etc/puppetlabs/code/environments` are created and owned by the Puppet user.

We can use Git's built-in sharedrepository setting to ensure that all members of the group have access to the repository, but the user's umask setting might prevent files from being created with group-write permissions. Putting a umask setting in our script and running Git using sudo is a more reliable way of ensuring access. To create a Git repository as a shared repository, use `shared=group` while creating the bare repository, as shown here:

```
git@stand$ cd /var/lib/git
git@stand$ git init --bare --shared=group newrepo.git
Initialized empty shared Git repository in /var/lib/git/newrepo.git/
```

First, we'll modify our `control.git` bare repository to enable shared access, and then we'll have to retroactively change the permissions to ensure that group access is granted. We'll edit `/var/lib/git/control.git/config`, as follows:

```
[core]
repositoryformatversion = 0
filemode = true
bare = true
sharedrepository = 1
```

To illustrate our workflow, we'll create a new group and add a user to that group, as shown here:

```
[root@stand ~]# groupadd pupdevs
[root@stand ~]# useradd -g pupdevs -c "Sample Developer" samdev [root@stand ~]# id samdev
uid=1002(samdev) gid=1002(pupdevs) groups=1002(pupdevs)
```

Now, we need to retroactively go back and change the ownership of files in `/var/lib/git/control.git` to ensure that the `pupdevs` group has write access to the repository. We'll also set the `setgid` bit on that directory so that new files are group owned by `pupdevs`, as shown here:

```
[root@stand ~]# cd /var/lib/git
[root@stand git]# find control.git -type d -exec chmod g+rwxs {} \;
[root@stand git]# find control.git -type f -exec chmod g+rw {} \;
[root@stand git]# chgrp -R pupdevs control.git
```

Now, the repository will be accessible to anyone in the `pupdevs` group. We now need to add a rule to our `sudoers` file to allow anyone in the `pupdevs` group to run Git as the Puppet user, using the following code:

```
%pupdevs ALL = (puppet) NOPASSWD: /bin/git *, /usr/bin/puppet-sync *
```

If your repo is still configured to use `puppet-sync` to push updates, then you need to remove the production directory from `/etc/puppetlabs/code/environments` before proceeding. `puppet-sync` creates a timestamp file (`.puppet-sync-stamp`) in the base of the directories it controls and will not update an existing directory by default.

With this sudo rule in place, sudo to `samdev`, clone the repository and modify the production branch, as shown:

```
[root@stand git]# sudo -iu samdev
[samdev@stand ~]$ git clone /var/lib/git/control.git/
Cloning into 'control'...
done.

[samdev@stand ~]$ cd control/
[samdev@stand control]$ git config --global user.name "Sample Developer"
[samdev@stand control]$ git config --global user.email "samdev@example.com"

[samdev@stand control]$ git checkout production
Branch production set up to track remote branch production from origin.
Switched to a new branch 'production'

[samdev@stand control]$ sed -i hieradata/hosts/client.yaml -e "s/welcome:.*welcome: 'Sample Developer made this change'/"

[samdev@stand control]$ echo "Example.com Puppet Control Repository" > README
[samdev@stand control]$ git add hieradata/hosts/client.yaml README
```

```
[samdev@stand control]$ git commit -m "Sample Developer changing welcome"
[production 49b7367] Sample Developer changing welcome
 2 files changed, 2 insertions(+), 1 deletion(-)
  create mode 100644 README
[samdev@stand control]$ git push origin production
Counting objects: 10, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 725 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To /var/lib/git/control.git/
    74d2ff5..49b7367  production -> production
```

We've updated our production branch. Our changes were automatically propagated to the Puppet environments directory. Now, we can run Puppet on a client (in the production environment) to see the changes, as shown:

```
[root@client ~]# puppet agent -t
Notice: /Stage[main]/Base/File[/etc/motd]/content:
--- /etc/motd      2015-10-01 23:59:39.289172885 -0700
+++ /tmp/puppet-file20151002-12893-1pbrr8a      2015-10-02
00:01:24.552178442 -0700
@@ -1,3 +1,3 @@
-Production Node: best behaviour.
+Sample Developer made this change
...
Notice: Applied catalog in 0.95 seconds
```

Now, any user we add to the pupdevs group will be able to update our Puppet code and have it pushed to any branch. If we look in /etc/puppetlabs/code/environments, we can see that the owner of the files is also the Puppet user due to the use of sudo, as shown here:

```
[samdev@stand ~]$ ls -l /etc/puppetlabs/code/environments
total 12
drwxr-xr-x. 6 root      root      86 Sep 26 19:46 master
drwxr-xr-x. 6 puppet    puppet    4096 Sep 26 22:02 production
drwxr-xr-x. 6 root      root      86 Sep 26 19:46 production.orig
drwxr-xr-x. 6 puppet    puppet    4096 Sep 26 21:42 puppet_sync
drwxr-xr-x. 6 puppet    puppet    4096 Sep 26 21:47 quiet
drwxr-xr-x. 6 puppet    puppet    86 Sep 26 20:48 thomas
```

Not playing nice with others via Git hooks

Our configuration at this point gives all users in the `pupdevs` group the ability to push changes to all branches. A usual complaint about Git is that it lacks a good system of access control. Using filesystem ACLs, it is possible to allow only certain users to push changes to specific branches. Another way to control commits is to use a pre-receive hook and verify if access will be granted before accepting the commit.

The pre-receive hook receives the same information as the post-receive hook. The hook runs as the user performing the commit so that we can use that information to block a user from committing to a branch or even doing certain types of commits. Merges, for instance, can be denied. To illustrate how this works, we'll create a new user called `newbie` and add them to the `pupdevs` group, using the following commands:

```
[root@stand ~]# useradd -g pupdevs -c "Rookie Developer" newbie
[root@stand ~]# sudo -iu newbie
```

We'll have `newbie` check our production code, make a commit, and then push the change to production, using the following commands:

```
[newbie@stand ~]$ git clone /var/lib/git/control.git
Cloning into 'control'...
done.

[newbie@stand ~]$ cd control
[newbie@stand control]$ git config --global user.name "Newbie"
[newbie@stand control]$ git config --global user.email "newbie@example.com"
[newbie@stand control]$ git checkout production
Branch production set up to track remote branch production from origin.
Switched to a new branch 'production'
[newbie@stand control]$ echo Rookie mistake >README
[newbie@stand control]$ git add README
[newbie@stand control]$ git commit -m "Rookie happens"
[production 23e0605] Rookie happens
 1 file changed, 1 insertion(+), 2 deletions(-)
```

Our rookie managed to wipe out the `README` file in the `production` branch. If this was an important file, then the deletion may have caused problems. It would be better if the rookie couldn't make changes to `production`. Note that this change hasn't been pushed up to the origin yet; it's only a local change.

We'll create a pre-receive hook that only allows certain users to commit to the production branch. Again, we'll use bash for simplicity. We will start by defining who will be allowed to commit and we are interested in protecting which branch, as shown in the following snippet:

```
#!/bin/bash

ALLOWED_USERS="samdev git root"
PROTECTED_BRANCH="production"
```

We will then use whoami to determine who has run the script (the developer who performed the commit), as follows:

```
user=$(whoami)
```

Now, just like we did in post-receive, we'll parse out the branch name and exit the script if we cannot determine the branch name, as shown in the following code:

```
read oldrev newrev refname
branch=${refname##*//*}
if [ -z $branch ]; then
    echo "ERROR: Branch undefined"
    exit 10
fi
```

We compare the \$branch variable against our protected branch and exit cleanly if this isn't a branch we are protecting, as shown in the following code. Exiting with an exit code of 0 informs Git that the commit should proceed:

```
if [ "$branch" != "$PROTECTED_BRANCH" ]; then
    # branch not protected, exit cleanly
    exit 0
fi
```

If we make it to this point in the script, we are on the protected branch and the \$user variable has our username. So, we will just loop through the \$ALLOWED_USERS variable looking for a user who is allowed to commit to the protected branch. If we find a match, we will exit cleanly, as shown in the following code:

```
for allowed in $ALLOWED_USERS
do
    if [ "$user" == "$allowed" ]; then
        # user allowed, exit cleanly
        echo "$PROTECTED_BRANCH change for $user"
        exit 0
    fi
done
```

If the user was not in the `$ALLOWED_USERS` variable, then their commit is denied and we exit with a non-zero exit code to inform Git that the commit should not be allowed, as shown in the following code:

```
# not an allowed user
echo "Error: Changes to $PROTECTED_BRANCH must be made by $ALLOWED_
USERS"
exit 10
```

Save this file with the name `pre-receive` in `/var/lib/git/puppet.git/hooks/` and then change the ownership to `git`. Make it executable using the following commands:

```
[root@stand ~]# chmod 755 /var/lib/git/control.git/hooks/pre-receive
[root@stand ~]# chown git:git /var/lib/git/control.git/hooks/pre-receive
```

Now, we'll go back and make a simple change to the repository as root. It is important to always get in the habit of running `git fetch` and `git pull origin <branch>` when you start working on a branch. You need to do this to ensure that you have the latest version of the branch from your origin:

```
[root@stand ~]# sudo -iu samdev
[samdev@stand ~]$ pwd
/home/samdev
[samdev@stand ~]$ ls
control
[samdev@stand ~]$ cd control
[samdev@stand control]$ git branch
  master
* production
[samdev@stand control]$ git fetch
[samdev@stand control]$ git pull origin production
From /var/lib/git/control
 * branch           production -> FETCH_HEAD
Already up-to-date.
[samdev@stand control]$ echo root >>README
[samdev@stand control]$ git add README
[samdev@stand control]$ git commit -m README
[production cd1be0f] README
 1 file changed, 1 insertion(+)
```

Now, with the simple changes made (we appended our username to the `README` file), we can push the change to the origin using the following command:

```
[samdev@stand control]$ git push origin production
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 324 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To /var/lib/git/control.git/
    b387c00..cd1be0f  production -> production
```

As expected, there are no errors and the `README` file is updated in the `production` branch by our post-receive hook. Now, we will attempt a similar change, as the `newbie` user. We haven't pushed our earlier change, so we'll try to push the change now, but first we have to merge the changes that `samdev` made by using `git pull`, as shown here:

```
[newbie@stand control]$ git pull origin production
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /var/lib/git/control
 * branch           production -> FETCH_HEAD
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

Our `newbie` user has wiped out the `README` file. They meant to append it to the file using two less than (`>>`) signs but instead used a single less than (`>`) sign and clobbered the file. Now, `newbie` needs to resolve the problems with the `README` file before they can attempt to push the change to `production`, as shown here:

```
[newbie@stand control]$ git add README
[newbie@stand control]$ git commit -m "fixing README"
[production 4ab787c] fixing README
```

Now `newbie` will attempt to push their changes up to the origin, as shown in the following example:

```
[newbie@stand control]$ git push origin production
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 324 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: ERROR: Changes to production must be made by samdev git root
To /var/lib/git/control.git
! [remote rejected] production -> production (pre-receive hook declined)
error: failed to push some refs to '/var/lib/git/control.git'
```

We see the commit beginning—the changes from the local production branch in `newbie` are sent to the origin. However, before working with the changes, Git runs the pre-receive hook and denies the commit. So, from the origin's perspective, the commit never took place. The commit only exists in the `newbie` user's directory. If the `newbie` user wishes this change to be propagated, he'll need to contact either `samdev`, `git`, or `root`.

Git for everyone

At this point, we've shown how to have Git work from one of the worker machines. In a real enterprise solution, the workers will have some sort of shared storage configured or another method of having Puppet code updated automatically. In that scenario, the Git repository wouldn't live on a worker but instead be pushed to a worker. Git has a workflow for this, which uses SSH keys to grant access to the repository. With minor changes to the shown solution, it is possible to have users SSH to a machine as the Git user to make commits.

First, we will have our developer generate an SSH key using the following commands:

```
[root@client ~]# sudo -iu remotedev
[remotedev@client ~]$ ssh-keygen
Generating public/private rsa key pair.
...
Your identification has been saved in /home/remotedev/.ssh/id_rsa.
Your public key has been saved in /home/remotedev/.ssh/id_rsa.pub.
The key fingerprint is:
18:52:85:e2:d7:cc:4d:b2:00:e1:5e:6b:25:70:ac:d6 remotedev@client.example.com
```

Then, copy the key into the `authorized_keys` file, for the Git user, as shown here:

```
remotedev@host $ ssh-copy-id -i ~/.ssh/id_rsa git@stand
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to
filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are
prompted now it is to install the new keys
```

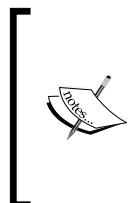
Number of key(s) added: 1

Now try logging into the machine, with `ssh 'git@stand'` and then check to make sure that only the key(s) you wanted were added:

```
[remotedev@client ~]$ ssh -i .ssh/id_rsa git@stand
Last login: Sat Sep 26 22:54:05 2015 from client
Welcome to Example.com
Managed Node: stand
Managed by Puppet version 4.2.1
```

If all is well, you should not be prompted for a password. If you are still being prompted for a password, check the permissions on `/var/lib/git` on the stand machine. The permissions should be 750 on the directory. Another issue may be SELinux security contexts; `/var/lib/git` is not a normal home directory location, so the contexts will be incorrect on the git user's `.ssh` directory. A quick way to fix this is to copy the context from the root user's `.ssh` directory, as shown here:

```
[root@stand git]# chcon -R --reference /root/.ssh .ssh
```



If you are copying the keys manually, remember that permissions are important here. They must be restrictive for SSH to allow access. SSH requires that `~git` (Git's home directory) should not be group writable, that `~git/.ssh` be 700, and also that `~git/.ssh/authorized_keys` be no more than 600. Check in `/var/log/secure` for messages from SSH if your remote user cannot SSH successfully as the Git user.

Git also ships with a restricted shell, `git-shell`, which can be used to only allow a user to update Git repositories. In our configuration, we will change the git user's shell to `git-shell` using `chsh`, as shown here:

```
[root@stand ~]# chsh -s $(which git-shell) git
Changing shell for git.
chsh: Warning: "/bin/git-shell" is not listed in /etc/shells.
Shell changed.
```

When a user attempts to connect to our machine as the `git` user, they will not be able to log in, as you can see here:

```
[remotedev@client ~]$ ssh -i .ssh/id_rsa git@stand
Last login: Sat Sep 26 23:13:39 2015 from client
Welcome to Example.com
Managed Node: stand
Managed by Puppet version 4.2.1
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to stand closed.
```

However, they will succeed if they attempted to use Git commands, as shown here:

```
[remotedev@client ~]$ git clone git@stand:control.git
Cloning into 'control'...
remote: Counting objects: 102, done.
remote: Compressing objects: 100% (71/71), done.
remote: Total 102 (delta 24), reused 0 (delta 0)
Receiving objects: 100% (102/102), 9.33 KiB | 0 bytes/s, done.
Resolving deltas: 100% (24/24), done.
```

Now, when a remote user executes a commit, it will run as the `git` user. We need to modify our `sudoers` file to allow sudo to run remotely. Add the following line at the top of `/etc/sudoers.d/sudoers-puppet` (possibly using visudo):

```
Defaults !requiretty
```

At this point, our sudo rule for the post-receive hook will work as expected, but we will lose the restrictiveness of our pre-receive hook since everything will be running as the `git` user. SSH has a solution to this problem, we can set an environment variable in the `authorized_keys` file that is the name of our remote user. Edit `~git/.ssh/authorized_keys`, as follows:

```
environment="USER=remotedev" ssh-rsa AAAA...b remotedev@client.
example.com
```

Finally, edit the pre-receive hook, by changing the `user=$(whoami)` line to `user=$USER`.

Now, when we use our SSH key to commit remotely, the environment variable set in the SSH key is used to determine who ran the commit.

Running an enterprise-level Git server is a complex task in itself. The scenario presented here can be used as a road map to develop your solution.

Summary

In this chapter, we have seen how to configure Puppet to work in different environments. We have seen how having `hieradata` in different environments can allow developers to work independently.

By leveraging the utility of Git and Git hooks, we can have custom-built environments for each developer, built automatically when the code is checked into our Git repository. This will allow us to greatly increase our developers' productivity and allow a team of system administrators to work simultaneously on the same code base.

In the next chapter, we'll see how public modules from Puppet Forge can be used to accomplish complex configurations on our nodes.

4

Public Modules

The default types shipped with Puppet can be used to do almost everything you need to do to configure your nodes. When you need to perform more tasks than the defaults can provide, you can either write your own custom modules or turn to the Forge (<http://forge.puppetlabs.com/>) and use a public module. Puppet Forge is a public repository of shared modules. Several of these modules enhance the functionality of Puppet, provide a new type, or solve a specific problem. In this chapter, we will first cover how to keep your public modules organized for your enterprise then we will go over specific use cases for some popular modules.

Getting modules

Modules are just files and a directory structure. They can be packaged as a ZIP archive or shared via a Git repository. Indeed, most modules are hosted on GitHub in addition to Puppet Forge. You will find most public modules on the Forge, and the preferred method to keep your modules up to date is to retrieve them from the Forge.

Using GitHub for public modules

If you have a module you wish to use and that is only hosted on GitHub (which is an online Git service for sharing code using Git), you can create a local Git repository and make the GitHub module a submodule of your modules. Another use for a local copy is if the public module does not work entirely as you require, you can modify the public module in your local copy.

This workflow has issues; submodules are local to each working copy of a module. When working in an enterprise, the internal servers do not usually have access to public Internet services such as GitHub. To get around this access problem, you can create an internal Git repository that is a clone of the public GitHub repository (the machine which is performing the clone operation will need to have access to GitHub).

We'll start by cloning the public repository as our `git` user:

```
[root@stand git]# sudo -u git bash
[git@stand ~]$ pwd
/var/lib/git
[git@stand ~]$ git clone --bare https://github.com/uphillian/
masteringpuppet.git
Cloning into bare repository 'masteringpuppet.git'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
```

We now have a local copy of our public repository. We'll create a checkout of this repository as our `remotedev` user on the client machine as shown here:

```
[remotedev@client ~]$ git clone git@stand:masteringpuppet.git
Cloning into 'masteringpuppet'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (4/4), 4.14 KiB | 0 bytes/s, done.
```

Now we'll create a local branch to track our internal changes to the module and name this branch `local`, as shown here:

```
[remotedev@client ~]$ cd masteringpuppet/
[remotedev@client masteringpuppet]$ git branch local
[remotedev@client masteringpuppet]$ git checkout local
Switched to branch 'local'
```

Now we will make our change to the local branch and add the modified `README.md` file to the repository. We then push our local branch back to the server (stand):

```
[remotedev@client masteringpuppet]$ git add README.md
[remotedev@client masteringpuppet]$ git commit -m "local changes"
[local 148ff2f] local changes
 1 file changed, 1 insertion(+), 1 deletion(-)
[remotedev@client masteringpuppet]$ git push origin local
Counting objects: 8, done.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 584 bytes | 0 bytes/s, done.
Total 6 (delta 1), reused 0 (delta 0)
To git@stand:masteringpuppet.git
 * [new branch]      local -> local
```

We now have a local branch that we can use internally. There are two issues with this configuration. When the public module is updated, we want to be able to pull those updates into our own module. We also want to be able to use our local branch wherever we want the module installed.

Updating the local repository

To pull in updates from the public module, you have to use the `git pull` command. First, add the public repository as a remote repository for our local clone as shown here:

```
[remotedev@client masteringpuppet]$ git remote add upstream git@github.com:uphillian/masteringpuppet.git
[remotedev@client masteringpuppet]$ git fetch upstream
```

The `git fetch` command is used to grab the latest data from the remote repository. With the latest version of the data available, we now use the `git pull` command to pull the latest changes into our current local branch as shown here:

```
[remotedev@client masteringpuppet]$ git pull upstream master
From github.com:uphillian/masteringpuppet
 * branch            master      -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 manifests/init.pp | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 manifests/init.pp
```

This will create an automatic merge of the upstream master branch in the local branch (provided there are no merge conflicts). Using the `git tag` command can be useful in this workflow. After each merge from the upstream public repository you can create a tag to refer to the current release of the repository. Using this local copy method we can use public modules within our organization without relying on direct connections to the public Internet. We are also able to make local modifications to our copy of the repository and maintain those changes independent of changes in the upstream module. This can become a problem if the upstream module makes changes that are incompatible with your changes. Any changes you make that can be pushed back to the upstream project are encouraged. Submitting a pull request on GitHub is a pain free way to share your improvements and modifications with the original developer.

Modules from the Forge

Modules on Puppet Forge can be installed using Puppet's built-in `module` command. The modules on the Forge have files named `Modulefile`, which define their dependencies; so, if you download modules from the Forge using `puppet module install`, then their dependencies will be resolved in a way similar to how `yum` resolves dependencies for `rpm` packages.

To install the `puppetlabs-puppetdb` module, we will simply issue a `puppet module install` command in the appropriate directory. We'll create a new directory in `tmp`; for our example, this will be `/tmp/public_modules`, as shown here:

```
[git@stand ~]$ cd /tmp  
[git@stand tmp]$ mkdir public_modules  
[git@stand tmp]$ cd public_modules/  
[git@stand public_modules]$
```

Then, we'll inform Puppet that our `modulepath` is `/tmp/public_modules` and install the `puppetdb` module using the following command:

```
[git@stand public_modules]$ puppet module install --modulepath=/tmp/  
public_modules puppetlabs-puppetdb  
Notice: Preparing to install into /tmp/public_modules ...  
Notice: Downloading from https://forgeapi.puppetlabs.com ...  
Notice: Installing -- do not interrupt ...  
/tmp/public_modules  
└─ puppetlabs-puppetdb (v5.0.0)  
    └─ puppetlabs-firewall (v1.7.1)
```

```

└── puppetlabs-inifile (v1.4.2)
    └── puppetlabs-postgresql (v4.6.0)
        ├── puppetlabs-apt (v2.2.0)
        ├── puppetlabs-concat (v1.2.4)
        └── puppetlabs-stdlib (v4.9.0)

```

Using `module install`, we retrieved `puppetlabs-firewall`, `puppetlabs-inifile`, `puppetlabs-postgresql`, `puppetlabs-apt`, `puppetlabs-concat`, and `puppetlabs-stdlib` all at once. So, not only have we satisfied dependencies automatically, but we also have retrieved release versions of the modules as opposed to the development code. We can, at this point, add these modules to a local repository and guarantee that our fellow developers will be using the same versions that we have checked out. Otherwise, we can inform our developers about the version we are using and have them check out the modules using the same versions.

You can specify the version with `puppet module install` as follows:

```

[git@stand public_modules]$ \rm -r stdlib
[git@stand public_modules]$ puppet module install --modulepath=/tmp/
public_modules puppetlabs-stdlib --version 4.8.0
Notice: Preparing to install into /tmp/public_modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/tmp/public_modules
└── puppetlabs-stdlib (v4.8.0)

```



The `\rm` in the previous example is a shorthand in UNIX to disable shell expansion of variables. `rm` is usually aliased to `rm -i`, which would have prompted us when we wanted to delete the directory.

Keeping track of the installed versions can become troublesome; a more stable approach is to use `librarian-puppet` to pull in the modules you require for your site.

Using Librarian

Librarian is a bundler for Ruby. It handles dependency checking for you. The project for using Librarian with Puppet is called `librarian-puppet` and is available at <http://rubygems.org/gems/librarian-puppet>. To install `librarian-puppet`, we'll use RubyGems since no rpm packages exist in public repositories at this time. To make our instructions platform agnostic, we'll use Puppet to install the package as shown here:

```
[root@stand ~]# puppet resource package librarian-puppet ensure=installed provider=gem
Notice: /Package[librarian-puppet]/ensure: created
package { 'librarian-puppet':
  ensure => ['2.2.1'],
}
```

We can now run `librarian-puppet` as follows:

```
[root@stand ~]# librarian-puppet version
librarian-puppet v2.2.1
```

The `librarian-puppet` project uses a `Puppetfile` to define the modules that will be installed. The syntax is the name of the module followed by a comma and the version to install. Modules may be pulled in from Git repositories or directly from Puppet Forge. You can override the location of Puppet Forge using a `forge` line as well. Our initial `Puppetfile` would be the following:

```
forge "http://forge.puppetlabs.com"
mod 'puppetlabs/puppetdb', '5.0.0'
mod 'puppetlabs/stdlib', '4.9.0'
```

We'll create a new public directory in `/tmp/public4` and include the `Puppetfile` in that directory, as shown here:

```
[git@stand ~]$ cd /tmp
[git@stand tmp]$ mkdir public4 && cd public4
[git@stand public4]$ cat<<EOF>Puppetfile
> forge "https://forgeapi.puppetlabs.com"
>mod 'puppetlabs/puppetdb', '5.0.0'
>mod 'puppetlabs/stdlib', '4.9.0'
> EOF
```

Next, we'll tell `librarian-puppet` to install everything we've listed in the `Puppetfile` as follows:

```
[git@stand public4]$ librarian-puppet update
[git@stand public4]$ ls
modules  Puppetfile  Puppetfile.lock
```

The `Puppetfile.lock` file is a file used by `librarian-puppet` to keep track of installed versions and dependencies; in our example, it contains the following:

```
FORGE
remote: https://forgeapi.puppetlabs.com
specs:
puppetlabs-apt (2.2.0)
puppetlabs-stdlib (< 5.0.0, >= 4.5.0)
puppetlabs-concat (1.2.4)
puppetlabs-stdlib (< 5.0.0, >= 3.2.0)
puppetlabs-firewall (1.7.1)
puppetlabs-inifile (1.4.2)
puppetlabs-postgresql (4.6.0)
puppetlabs-apt (< 3.0.0, >= 1.8.0)
puppetlabs-concat (< 2.0.0, >= 1.1.0)
puppetlabs-stdlib (~> 4.0)
puppetlabs-puppetdb (5.0.0)
puppetlabs-firewall (< 2.0.0, >= 1.1.3)
puppetlabs-inifile (< 2.0.0, >= 1.1.3)
puppetlabs-postgresql (< 5.0.0, >= 4.0.0)
puppetlabs-stdlib (< 5.0.0, >= 4.2.2)
puppetlabs-stdlib (4.9.0)

DEPENDENCIES
puppetlabs-puppetdb (= 5.0.0)
puppetlabs-stdlib (= 4.9.0)
```

Our modules are installed in `/tmp/public4/modules`. Now, we can go back and add all these modules to our initial `Puppetfile` to lockdown the versions of the modules for all our developers. The process for a developer to clone our working tree would be to install `librarian-puppet` and then pull down our `Puppetfile`. We will add the `Puppetfile` to our Git repository to complete the workflow. Thus, each developer will be guaranteed to have the same public module structure.

We can then move these modules to `/etc/puppetlabs/code/modules` and change permissions for the Puppet user using the following commands:

```
[root@stand ~]# cd /tmp/public4/modules/
[root@stand modules]# cp -a . /etc/puppetlabs/code/modules/
[root@stand modules]# chown -R puppet:puppet /etc/puppetlabs/code/modules
[root@stand modules]# ls /etc/puppetlabs/code/modules/
apt concat firewall inifile postgresql puppetdb stdlib
```

This method works fairly well, but we still need to update the modules independently of our Git updates; we need to do these two actions together. This is where `r10k` comes into play.

Using r10k

`r10k` is an automation tool for Puppet environments. It is hosted on GitHub at <https://github.com/puppetlabs/r10k>. The project is used to speed up deployments when there are many environments and many Git repositories in use. From what we've covered so far, we can think of it as `librarian-puppet` and Git hooks in a single package. `r10k` takes the Git repositories specified in `/etc/puppetlabs/r10k/r10k.yaml` and checks out each branch of the repositories into a subdirectory of the environment directory (the environment directory is also specified in `/etc/puppetlabs/r10k/r10k.yaml`). If there is a `Puppetfile` in the root of the branch, then `r10k` parses the file in the same way that `librarian-puppet` does and it installs the specified modules in a directory named `modules` under the environment directory.

To use `r10k`, we'll replace our `post-receive` Git hook from the previous chapter with a call to `r10k` and we'll move our `librarian-puppet` configuration to a place where `r10k` is expecting it. We'll be running `r10k` as the `puppet` user, so we'll set up the `puppet` user with a normal shell and login files, as shown here:

```
[root@stand ~]# chsh -s /bin/bash puppet
Changing shell for puppet.
Shell changed.
[root@stand ~]# cp /etc/skel/.bash* ~puppet/
[root@stand ~]# chown puppet ~puppet/.bash*
[root@stand ~]# sudo -iu puppet
[puppet@stand ~]$
```

Now, install the r10k gem as shown here:

```
[root@stand ~]# puppet resource package r10k ensure=present provider=gem
Notice: /Package[r10k]/ensure: created
package { 'r10k':
  ensure => ['2.0.3'],
}
```

Next, we'll create a /etc/puppetlabs/r10k/r10k.yaml file to point to our local Git repository. We will also specify that our Puppet environments will reside in /etc/puppetlabs/code/environments, as shown in the following snippet:

```
---
cachedir: '/var/cache/r10k'
sources:
control:
remote: '/var/lib/git/control.git'
basedir: '/etc/puppetlabs/code/environments'
```

Now, we need to create the cache directory and make it owned by the puppet user. We will use the following commands to do so:

```
[root@stand ~]# mkdir /var/cache/r10k
[root@stand ~]# chown puppet:puppet /var/cache/r10k
```

Now, we need to check out our code and add a Puppetfile to the root of the checkout. In each environment, create a Puppetfile that contains which modules you want installed in that environment; we'll copy the previous Puppetfile as shown in the following code:

```
forge "http://forge.puppetlabs.com"
mod 'puppetlabs/puppetdb', '5.0.0'
mod 'puppetlabs/stdlib', '4.9.0'
```

We'll check the syntax of our Puppetfile using r10k as shown here:

```
[samdev@stand control]$ cat Puppetfile
forge "http://forge.puppetlabs.com"
mod 'puppetlabs/puppetdb', '5.0.0'
mod 'puppetlabs/stdlib', '4.9.0'
[samdev@stand control]$ r10k puppetfile check
Syntax OK
```

Now, add the Puppetfile to the Git repository using the following commands:

```
[samdev@stand control]$ git add Puppetfile
[samdev@stand control]$ git commit -m "adding Puppetfile"
[production 17d53ad] adding Puppetfile
 1 file changed, 3 insertions(+)
create mode 100644 Puppetfile
```

Now, r10k expects that the modules specified in the Puppetfile will get installed in \$environment/modules, but we already have modules in that location. Move the existing modules into another directory using the following commands; dist or local are commonly used:

```
[samdev@stand control]$ git mv modules dist
[samdev@stand control]$ git commit -m "moving modules to dist"
[production d3909a3] moving modules to dist
 6 files changed, 0 insertions(+), 0 deletions(-)
 rename {modules => dist}/base/manifests/init.pp (100%)
 rename {modules => dist}/hostname_problem/manifests/init.pp (100%)
 rename {modules => dist}/resolver/manifests/init.pp (100%)
 rename {modules => dist}/syslog/manifests/init.pp (100%)
 rename {modules => dist}/virtual/manifests/init.pp (100%)
 rename {modules => dist}/web/manifests/init.pp (100%)
```

Now that our modules are out of the way, we don't want a modules directory to be tracked by Git, so add modules to .gitignore using the following commands:

```
[samdev@stand control]$ echo "modules/" >>.gitignore
[samdev@stand control]$ git add .gitignore
[samdev@stand control]$ git commit -m "adding .gitignore"
[production e6a5a4a] adding .gitignore
 1 file changed, 1 insertion(+)
create mode 100644 .gitignore
```

Ok, we are finally ready to test. Well almost. We want to test r10k, so we need to disable our post-receive hook; just disable the execute bit on the script using the following commands:

```
[root@stand ~]# sudo -u git bash
[git@stand ~]$ cd /var/lib/git/control.git/hooks
[git@stand hooks]$ chmod -x post-receive
```

Now we can finally add our changes to the Git repository, using the following commands:

```
[git@stand hooks]$ exit
exit
[root@stand ~]# sudo -iu samdev
[samdev@stand ~]$ cd control
[samdev@stand control]$ git push origin production
Counting objects: 9, done.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (8/8), 946 bytes | 0 bytes/s, done.
Total 8 (delta 2), reused 0 (delta 0)
remote: production
remote: production change for samdev
To /var/lib/git/control.git/
  0d5cf62..e6a5a4a  production -> production
```

Note that the only remote lines in the output are related to our pre-receive hook since we no longer have a post-receive hook running.

We will be running r10k as the puppet user, so we'll need to ensure that the puppet user can access files in the /var/lib/git directory; we'll use **Filesystem Access Control Lists (FACLs)** to achieve this access as shown here:

```
[root@stand ~]# setfacl -m 'g:puppet:rwx' -R /var/lib/git
[root@stand ~]# setfacl -m 'g:puppet:rwx' -R -d /var/lib/git
```

Before we can use r10k, we need to clean out the environments directory using the following commands:

```
[samdev@stand control]$ exit
logout
[root@stand ~]# sudo chown puppet /etc/puppetlabs/code
[root@stand ~]# sudo -iu puppet
[puppet@stand ~]$ cd /etc/puppetlabs/code
[puppet@stand code]$ mv environments environments.b4-r10k
[puppet@stand code]$ mkdir environments
```

Now we can test r10k using r10k deploy as follows:

```
[puppet@stand code]$ r10k deploy environment -p
[puppet@stand code]$ ls environments
master  production  puppet_sync  quiet  thomas
```

As we can see, r10k did a Git checkout of our code in the `master`, `thomas`, `quiet`, and `production` branches. We added a `Puppetfile` to the `production` branch; so, when we look in `/etc/puppetlabs/code/environments/production/modules`, we will see the `puppetdb` and `stdlib` modules defined in the `Puppetfile`:

```
[puppet@stand code]$ ls environments/production/modules
puppetdb  stdlib
```

We have now used r10k to deploy not only our code but the `puppetdb` and `stdlib` modules as well. We'll now switch our workflow to use r10k and change our post-receive hook to use r10k. Our post-receive hook will be greatly simplified; we'll just call r10k with the name of the branch and exit. Alternatively, we can have r10k run on every environment if we choose to; this way, it will only update a specific branch each time. To make the hook work again, we'll first need to enable the execute bit on the file, using the following commands:

```
[root@stand ~]# sudo -u git bash
[git@stand root]$ cd /var/lib/git/control.git/hooks/
[git@stand hooks]$ chmod +x post-receive
```

Next, we'll replace the contents of `post-receive` with the following script:

```
logout
#!/bin/bash
r10k=/usr/local/bin/r10k
read oldrev newrev refname
branch=${refname##*//*\/}
if [ -z "$branch" ]; then
    echo "ERROR: Branch undefined"
    exit 10
fi

exec sudo -u puppet $r10k deploy environment $branch -p
```

Now, we need to edit our `sudoers` file to allow Git to run r10k as `puppet`, as shown here:

```
Defaults !requiretty
git ALL = (puppet) NOPASSWD: /bin/git *, /usr/bin/puppet-sync *, /usr/
local/bin/r10k *
%pupdevs ALL = (puppet) NOPASSWD: /bin/git *, /usr/bin/puppet-sync *,
/usr/local/bin/r10k *
```

Now, to test whether everything is working, remove a module from the production environment using the following command:

```
[root@stand ~]# \rm -rf /etc/puppetlabs/code/environments/production/
modules/stdlib
```

Now, make a change in production and push that change to the origin to trigger an r10k run, as shown here:

```
[root@stand ~]# sudo -iu samdev
[samdev@stand ~]$ cd control/
[samdev@stand control]$ echo "Using r10k in post-receive" >>README
[samdev@stand control]$ git add README
[samdev@stand control]$ git commit -m "triggering r10k rebuild"
[production bab33bd] triggering r10k rebuild
 1 file changed, 1 insertion(+)
[samdev@stand control]$ git push origin production
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 295 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: production
remote: production change for samdev
To /var/lib/git/control.git/
 422de2d..bab33bd  production -> production
```

Finally, verify whether the stdlib module was recreated or not using the following command:

```
[samdev@stand control]$ ls /etc/puppetlabs/code/environments/production/
modules/
puppetdb  stdlib
```

Keeping everything in r10k allows us to have mini labs for developers to work on a copy of our entire infrastructure with a few commands. They will only need a copy of our Git repository and our `r10k.yaml` file to recreate the configuration on a private Puppet master.

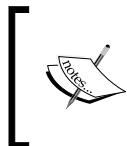
Using Puppet-supported modules

Many of the modules found on the public Forge are of high quality and have good documentation. The modules we will cover in this section are well-documented. What we will do is use concrete examples to show how to use these modules to solve real-world problems. Though I have covered only those modules I personally found useful, there are many excellent modules that can be found on the Forge. I encourage you to have a look at them first before starting to write your own modules.

The modules that we will cover are as follows:

- concat
- ini_file
- firewall
- lvm
- stdlib

These modules extend Puppet with custom types and, therefore, require that `pluginsync` be enabled on our nodes. `pluginsync` copies Ruby libraries from the modules to `/opt/puppetlabs/puppet/cache/lib/puppet` and `/opt/puppetlabs/puppet/cache/lib/facter`.



`pluginsync` is enabled by default in Puppet versions 3.0 and higher; no configuration is required. If you are using a version prior to 3.0, you will need to enable `pluginsync` in your `puppet.conf`.

concat

When we distribute files with Puppet, we either send the whole file as is or we send over a template that has references to variables. The `concat` module offers us a chance to build up a file from fragments and have it reassembled on the node. Using `concat`, we can have files which live locally on the node incorporated into the final file as sections. More importantly, while working in a complex system, we can have more than one module adding sections to the file. In a simple example, we can have four modules, all of which operate on `/etc/issue`. The modules are as follows:

- `issue`: This is the base module that puts a header on `/etc/issue`
- `issue_confidential`: This module adds a confidential warning to `/etc/issue`
- `issue_secret`: This module adds a secret level warning to `/etc/issue`
- `issue_topsecret`: This module adds a top secret level warning to `/etc/issue`

Using either the file or the template method to distribute the file won't work here because all of the four modules are modifying the same file. What makes this harder still is that we will have machines in our organization that require one, two, three, or all four of the modules to be applied. The concat module allows us to solve this problem in an organized fashion (not a haphazard series of execs with awk and sed). To use concat, you first define the container, which is the file that will be populated with the fragments. concat calls the sections of the file fragments. The fragments are assembled based on their order. The order value is assigned to the fragments and should have the same number of digits, that is, if you have 100 fragments then your first fragment should have 001, and not 1, as the order value. Our first module issue will have the following `init.pp` manifest file:

```
class issue {
    concat { 'issue':
        path => '/etc/issue',
    }
    concat::fragment {'issue_top':
        target  => 'issue',
        content => "Example.com\n",
        order   => '01',
    }
}
```

This defines `/etc/issue` as a concat container and also creates a fragment to be placed at the top of the file (order01). When applied to a node, the `/etc/issue` container will simply contain `Example.com`.

Our next module is `issue_confidential`. This includes the `issue` module to ensure that the container for `/etc/issue` is defined and we have our header. We then define a new fragment to contain the confidential warning, as shown in the following code:

```
class issue_confidential {
    include issue
    concat::fragment {'issue_confidential':
        target  => 'issue',
        content => "Unauthorised access to this machine is strictly
                    prohibited. Use of this system is limited to authorised
                    parties only.\n",
        order   => '05',
    }
}
```

This fragment has `order 5`, so it will always appear after the header. The next two modules are `issue_secret` and `issue_topsecret`. They both perform the same function as `issue_confidential` but with different messages and orders, as you can see in the following code:

```
class issue_secret {
  include issue
  concat::fragment {'issue_secret':
    target  => 'issue',
    content => "All information contained on this system is
                protected, no information may be removed from the system
                unless authorised.\n",
    order   => '10',
  }
}
class issue_topsecret {
  include issue
  concat::fragment {'issue_topsecret':
    target  => 'issue',
    content => "You should forget you even know about this
                system.\n",
    order   => '15',
  }
}
```

We'll now add all these modules to the control repository in the `dist` directory. We also update the `Puppetfile` to include the location of the `concat` module, as shown here:

```
mod 'puppetlabs(concat'
```

We next need to update our `environment.conf` file to include the `dist` directory as shown here:

```
modulepath = modules:$basemodulepath:dist
```

Using our Hiera configuration from the previous chapter, we will modify the `client.yaml` file to contain the `issue_confidential` class as shown here:

```
---
welcome: 'Sample Developer made this change'
classes:
  - issue_confidential
```

This configuration will cause the `/etc/issue` file to contain the header and the confidential warning. To have these changes applied to our `/etc/puppetlabs/code/environments` directory by r10k, we'll need to add all the files to the Git repository and push the changes, as shown here:

```
[samdev@stand control]$ git status
# On branch production
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   Puppetfile
#       new file:   dist/issue/manifests/init.pp
#       new file:   dist/issue_confidential/manifests/init.pp
#       new file:   dist/issue_secret/manifests/init.pp
#       new file:   dist/issue_topsecret/manifests/init.pp
#       modified:   environment.conf
#       modified:   hieradata/hosts/client.yaml
#
[samdev@stand control]$ git commit -m "concat example"
[production 6b3e7ae] concat example
 7 files changed, 39 insertions(+), 1 deletion(-)
create mode 100644 dist/issue/manifests/init.pp
create mode 100644 dist/issue_confidential/manifests/init.pp
create mode 100644 dist/issue_secret/manifests/init.pp
create mode 100644 dist/issue_topsecret/manifests/init.pp
[samdev@stand control]$ git push origin production
Counting objects: 27, done.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (20/20), 2.16 KiB | 0 bytes/s, done.
Total 20 (delta 2), reused 0 (delta 0)
remote: production
remote: production change for samdev
To /var/lib/git/control.git/
  bab33bd..6b3e7ae  production -> production
```

Since concat was defined in our Puppetfile, we will now see the concat module in /etc/puppetlabs/code/environments/production/modules as shown here:

```
[samdev@stand control]$ ls /etc/puppetlabs/code/environments/production/
modules/
concat  puppetdb  stdlib
```

We are now ready to run Puppet agent on client, after a successful Puppet agent run we see the following while attempting to log in to the system:

```
Example.com
Unauthorised access to this machine is strictly
prohibited. Use of this system is limited to authorized
parties only.
client login:
```

Now, we will go back to our client.yaml file and add issue_secret, as shown in the following snippet:

```
---
welcome: 'Sample Developer Made this change'
classes: - issue_confidential
         - issue_secret
```

After a successful Puppet run, the login looks like the following:

```
Example.com
Unauthorised access to this machine is strictly
prohibited. Use of this system is limited to authorised
parties only.
All information contained on this system is protected, no information may
be removed from the system unless authorised.
client login:
```

Adding the issue_topsecret module is left as an exercise, but we can see the utility of being able to have several modules modify a file. We can also have a fragment defined from a file on the node. We'll create another module called issue_local and add a local fragment. To specify a local file resource, we will use the source attribute of concat::fragment, as shown in the following code:

```
class issue_local {
  include issue
  concat::fragment {'issue_local':
    target  => 'issue',
```

```
    source => '/etc/issue.local',
    order   => '99',
}
}
```

Now, we add `issue_local` to `client.yaml`, but before we can run Puppet agent on client, we have to create `/etc/issue.local`, or the catalog will fail. This is a shortcoming of the `concat` module—if you specify a local path, then it has to exist. You can overcome this by having a file resource defined that creates an empty file if the local path doesn't exist, as shown in the following snippet:

```
file {'issue_local':
  path => '/etc/issue.local',
  ensure => 'file',
}
}
```

Then, modify the `concat::fragment` to require the file resource, as shown in the following snippet:

```
concat::fragment {'issue_local':
  target  => 'issue',
  source   => '/etc/issue.local',
  order    => '99',
  require  => File['issue_local'],
}
}
```

Now, we can run Puppet agent on `node1`; nothing will happen but the catalog will compile. Next, add some content to `/etc/issue.local` as shown here:

```
node1# echo "This is an example node, avoid storing protected material
here" >/etc/issue.local
```

Now after running Puppet, our login prompt will look like this:

```
Example.com
Unauthorised access to this machine is strictly
prohibited. Use of this system is limited to authorised
parties only.

All information contained on this system is protected, no information may
be removed from the system unless authorised.

This is an example node, avoid storing protected material here
client login:
```

There are many places where you would like to have multiple modules modify a file. When the structure of the file isn't easily determined, `concat` is the only viable solution. If the file is highly structured, then other mechanisms such as `augeas` can be used. When the file has a syntax of the `inifile` type, there is a module specifically made for `inifiles`.

inifile

The `inifile` module modifies the ini-style configuration files, such as those used by Samba, **System Security Services Daemon (SSSD)**, yum, tuned, and many others, including Puppet. The module uses the `ini_setting` type to modify settings based on their section, name, and value. We'll add `inifile` to our `Puppetfile` and push the change to our production branch to ensure that the `inifile` module is pulled down to our client node on the next Puppet agent run. Begin by adding the `inifile` to the `Puppetfile` as shown here:

```
mod 'puppetlabs/inifile'
```

With the module in the `Puppetfile` and pushed to the repository, r10k will download the module as we can see from the listing of the `production/modules` directory:

```
[samdev@stand control]$ ls/etc/puppetlabs/code/environments/production/
modules/
concat  inifile  puppetdb  stdlib
```

To get started with `inifile`, we'll look at an example in the `yum.conf` configuration file (which uses ini syntax). Consider the `gpgcheck` setting in the following `/etc/yum.conf` file:

```
[main]
cachedir=/var/cache/yum/$basearch/$releasever
keepcache=0
debuglevel=2
logfile=/var/log/yum.log
exactarch=1
obsoletes=1
gpgcheck=1
plugins=1
installonly_limit=3
```

As an example, we will modify that setting using `puppet resource`, as shown here:

```
[root@client ~]# puppet resource ini_setting dummy_name path=/etc/yum.conf section=main setting=gpgcheck value=0
Notice: /Ini_setting[dummy_name]/value: value changed '1' to '0'
ini_setting { 'dummy_name':
  ensure => 'present',
  value  => '0',
}
```

When we look at the file, we will see that the value was indeed changed:

```
[main]
cachedir=/var/cache/yum/$basearch/$releasever
keepcache=0
debuglevel=2
logfile=/var/log/yum.log
exactarch=1
obsoletes=1
gpgcheck=0
```

The power of this module is the ability to change only part of a file and not clobber the work of another module. To show how this can work, we'll modify the SSSD configuration file. SSSD manages access to remote directories and authentication systems. It supports talking to multiple sources; we can exploit this to create modules that only define their own section of the configuration file. In this example, we'll assume there are production and development authentication LDAP directories called `prod` and `devel`. We'll create modules called `sssd_prod` and `sssd-devel` to modify the configuration file. Starting with `sssd`, we'll create a module which creates the `/etc/sssd` directory:

```
class sssd {
  file { '/etc/sssd':
    ensure => 'directory',
    mode   => '0755',
  }
}
```

Next we'll create `sssd_prod` and add a `[domain/PROD]` section to the file, as shown in the following snippet:

```
class sssd_prod {
  include sssd
  Ini_setting { require => File['/etc/sssd'] }
  ini_setting {'krb5_realm_prod':
```

```
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/PROD',
    setting   => 'krb5_realm',
    value     => 'PROD',
}
ini_setting {'ldap_search_base_prod':
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/PROD',
    setting   => 'ldap_search_base',
    value     => 'ou=prod,dc=example,dc=com',
}
ini_setting {'ldap_uri_prod':
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/PROD',
    setting   => 'ldap_uri',
    value     => 'ldaps://ldap.prod.example.com',
}
ini_setting {'krb5_kpasswd_prod':
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/PROD',
    setting   => 'krb5_kpasswd',
    value     => 'secret!',
}
ini_setting {'krb5_server_prod':
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/PROD',
    setting   => 'krb5_server',
    value     => 'kdc.prod.example.com',
}
```

These `ini_setting` resources will create five lines within the `[domain/PROD]` section of the configuration file. We need to add `PROD` to the list of domains; for this, we'll use `ini_subsetting` as shown in the following snippet. The `ini_subsetting` type allows us to add sub settings to a single setting:

```
ini_subsetting {'domains_prod':
    path      => '/etc/sssd/sssd.conf',
    section   => 'sssd',
    setting   => 'domains',
    subsetting => 'PROD',
}
```

Now, we'll add `sssd_prod` to our `client.yaml` file and run `puppet agent -t` on client to see the changes, as shown here:

```
[root@client ~]# puppet agent -t
...
Info: Applying configuration version '1443519502'
Notice: /Stage[main]/Sssd_prod/File[/etc/sssd]/ensure: created
...
Notice: /Stage[main]/Sssd_prod/Ini_setting[krb5_server_prod]/ensure: created
Notice: /Stage[main]/Sssd_prod/Ini_subsetting[domains_prod]/ensure: created
Notice: Applied catalog in 1.07 seconds
```

Now when we look at `/etc/sssd/sssd.conf`, we will see the `[sssd]` and `[domain/PROD]` sections are created (they are incomplete for this example; you will need many more settings to make SSSD work properly), as shown in the following snippet:

```
[sssd]
domains = PROD

[domain/PROD]
krb5_server = kdc.prod.example.com
krb5_kpasswd = secret!
ldap_search_base = ou=prod,dc=example,dc=com
ldap_uri = ldaps://ldap.prod.example.com
krb5_realm = PROD
```

Now, we can create our `sssd-devel` module and add the same setting as that we did for `PROD`, changing their values for `DEVEL`, as shown in the following code:

```
class sssd-devel {
  include sssd
  Ini_setting { require => File['/etc/sssd'] }
  ini_setting {'krb5_realm-devel':
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/DEVEL',
    setting   => 'krb5_realm',
    value     => 'DEVEL',
  }
  ini_setting {'ldap_search_base-devel':
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/DEVEL',
    setting   => 'ldap_search_base',
```

```
        value    => 'ou=devel,dc=example,dc=com',
    }
ini_setting {'ldap_uri-devel':
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/DEVEL',
    setting   => 'ldap_uri',
    value     => 'ldaps://ldap.devel.example.com',
}
ini_setting {'krb5_kpasswd-devel':
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/DEVEL',
    setting   => 'krb5_kpasswd',
    value     => 'DevelopersDevelopersDevelopers',
}
ini_setting {'krb5_server-devel':
    path      => '/etc/sssd/sssd.conf',
    section   => 'domain/DEVEL',
    setting   => 'krb5_server',
    value     => 'dev1.devel.example.com',
}
```

Again, we will add DEVEL to the list of domains using `ini_subsetting`, as shown in the following code:

```
ini_subsetting {'domains-devel':
    path      => '/etc/sssd/sssd.conf',
    section   => 'sssd',
    setting   => 'domains',
    subsetting => 'DEVEL',
}
```

Now, after adding `sssd-devel` to `client.yaml`, we run Puppet agent on client and examine the `/etc/sssd/sssd.conf` file after, which is shown in the following snippet:

```
[sssd]
domains = PROD DEVEL

[domain/PROD]
krb5_server = kdc.prod.example.com
krb5_kpasswd = secret!
ldap_search_base = ou=prod,dc=example,dc=com
ldap_uri = ldaps://ldap.prod.example.com
krb5_realm = PROD
```

```
[domain/DEVEL]
krb5_realm = DEVEL
ldap_uri = ldaps://ldap.devel.example.com
ldap_search_base = ou=devel,dc=example,dc=com
krb5_server = dev1.devel.example.com
krb5_kpasswd = DevelopersDevelopersDevelopers
```

As we can see, both realms have been added to the `domains` section and each realm has had its own configuration section created. To complete this example, we will need to enhance the `SSSD` module that each of these modules calls with `include sssd`. In that module, we will define the `SSSD` service and have our changes send a `notify` signal to the service. I would place the `notify` signal in the domain's `ini_subsetting` resource.

Having multiple modules work on the same files simultaneously can make your Puppet implementation a lot simpler. It's counterintuitive, but having the modules coexist means you don't need as many exceptions in your code. For example, the `Samba` configuration file can be managed by a `Samba` module, but shares can be added by other modules using `inifile` and not interfere with the main `Samba` module.

firewall

If your organization uses host-based firewalls, filters that run on each node filtering network traffic, then the `firewall` module will soon become a friend. On enterprise Linux systems, the `firewall` module can be used to configure `iptables` automatically. Effective use of this module requires having all your `iptables` rules in Puppet.



The `firewall` module has some limitations—if your systems require large rulesets, your agent runs may take some time to complete. EL7 systems use `firewalld` to manage `iptables`, `firewalld` is not supported by the `firewall` module. Currently, this will cause execution of the following code to error on EL7 systems, but the `iptables` rules will be modified as expected.

The default configuration can be a little confusing; there are ordering issues that have to be dealt with while working with the firewall rules. The idea here is to ensure that there are no rules at the start. This is achieved with `purge`, as shown in the following code:

```
resources { "firewall":
  purge => true
}
```

Next, we need to make sure that any firewall rules we define are inserted after our initial configuration rules and before our final deny rule. To ensure this, we use a resource default definition. Resource defaults are made by capitalizing the resource type. In our example, `firewall` becomes `Firewall`, and we define the `before` and `require` attributes such that they point to the location where we will keep our setup rules (`pre`) and our final deny statement (`post`), as shown in the following snippet:

```
Firewall {
  before => Class['example_fw::post'],
  require => Class['example_fw::pre'],
}
```

Because we are referencing `example_fw::pre` and `example_fw::post`, we'll need to include them at this point. The module also defines a `firewall` class that we should include. Rolling all that together, we have our `example_fw` class as the following:

```
class example_fw {
  include example_fw::post
  include example_fw::pre
  include firewall

  resources { "firewall":
    purge => true
  }
  Firewall {
    before => Class['example_fw::post'],
    require => Class['example_fw::pre'],
  }
}
```

Now we need to define our default rules to go to `example_fw::pre`. We will allow all ICMP traffic, all established and related TCP traffic, and all SSH traffic. Since we are defining `example_fw::pre`, we need to override our earlier `require` attribute at the beginning of this class, as shown in the following code:

```
class example_fw::pre {
  Firewall {
    require => undef,
  }
}
```

Then, we can add our rules using the `firewall` type provided by the module. When we define the `firewall` resources, it is important to start the name of the resource with a number, as shown in the following snippet. The numbers are used for ordering by the `firewall` module:

```
firewall { '000 accept all icmp':
  proto => 'icmp',
```

```
    action => 'accept',
}
firewall { '001 accept all to lo':
    proto => 'all',
    iniface => 'lo',
    action => 'accept',
}
firewall { '002 accept related established':
    proto => 'all',
    state => ['RELATED', 'ESTABLISHED'],
    action => 'accept',
}
firewall { '022 accept ssh':
    proto => 'tcp',
    dport => '22',
    action => 'accept',
}
```

Now, if we finished at this point, our rules would be a series of `allow` statements. Without a final `deny` statement, everything is allowed. We need to define a `drop` statement in our `post` class. Again, since this is `example_fw::post`, we need to override the earlier setting to `before`, as shown in the following code:

```
class example_fw::post {
    firewall { '999 drop all':
        proto => 'all',
        action => 'drop',
        before => undef,
    }
}
```

Now, we can apply this class in our `node1.yaml` file and run Puppet to see the firewall rules getting rewritten by our module. The first thing we will see is the current firewall rules being purged.

Next, our `pre` section will apply our initial `allow` rules:

```
Notice: /Stage[main]/Example_fw::Pre/Firewall[002 accept related
established]/ensure: created
Notice: /Stage[main]/Example_fw::Pre/Firewall[000 accept all icmp]/
ensure: created
Notice: /Stage[main]/Example_fw::Pre/Firewall[022 accept ssh]/ensure:
created
Notice: /Stage[main]/Example_fw::Pre/Firewall[001 accept all to lo]/
ensure: created
```

Finally, our post section adds a drop statement to the end of the rules, as shown here:

```
Notice: /Stage[main]/Example_fw::Post/Firewall[999 drop all]/ensure:
created
Notice: Finished catalog run in 5.90 seconds
```

Earlier versions of this module did not save the rules; you would need to execute `iptables-save` after the post section. The module now takes care of this so that when we examine `/etc/sysconfig/iptables`, we see our current rules saved, as shown in the following snippet:

```
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [1:180]
-A INPUT -p icmp -m comment --comment "000 accept all icmp" -j ACCEPT
-A INPUT -i lo -m comment --comment "001 accept all to lo" -j ACCEPT
-A INPUT -m comment --comment "002 accept related established" -m
state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -p tcp -m multiport --dports 22 -m comment --comment "022
accept ssh" -j ACCEPT
-A INPUT -m comment --comment "999 drop all" -j DROP
COMMIT
```

Now that we have our firewall controlled by Puppet, when we apply our web module to our node, we can have it open port 80 on the node as well, as shown in the following code. Our earlier web module can just use `include example_fw` and define a `firewall` resource:

```
class web {
  package {'httpd':
    ensure => 'installed'
  }
  service {'httpd':
    ensure  => true,
    enable  => true,
    require => Package['httpd'],
  }
  include example_fw
  firewall {'080 web server':
    proto  => 'tcp',
    port   => '80',
    action  => 'accept',
  }
}
```

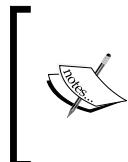
Now when we apply this class to an EL6 node, `e16`, we will see that port 80 is applied after our SSH rule and before our deny rule as expected:

```
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [1:164]
-A INPUT -p icmp -m comment --comment "000 accept all icmp" -j ACCEPT
-A INPUT -i lo -m comment --comment "001 accept all to lo" -j ACCEPT
-A INPUT -m comment --comment "002 accept related established" -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -p tcp -m multiport --dports 22 -m comment --comment "022 accept ssh" -j ACCEPT
-A INPUT -p tcp -m multiport --dports 80 -m comment --comment "080 web server" -j ACCEPT
-A INPUT -m comment --comment "999 drop all" -j DROP
COMMIT
```

Using this module, it's possible to have very tight host-based firewalls on your systems that are flexible and easy to manage.

Logical volume manager

The logical volume manager module allows you to create volume groups, logical volumes, and filesystems with Puppet using the **logical volume manager (lvm)** tools in Linux.



Having Puppet automatically configure your logical volumes can be a great benefit, but it can also cause problems. The module is very good at not shrinking filesystems, but you may experience catalog failures when physical volumes do not have sufficient free space.

If you are not comfortable with LVM, then I suggest you do not start with this module. This module can be of great help if you have products that require their own filesystems or auditing requirements that require application logs to be on separate filesystems. The only caveat here is that you need to know where your physical volumes reside, that is, which device contains the physical volumes for your nodes. If you are lucky and have the same disk layout for all nodes, then creating a new filesystem for your audit logs, `/var/log/audit`, is very simple. Assuming that we have an empty disk at `/dev/sdb`, we can create a new volume group for audit items and a logical volume to contain our filesystem. The module takes care of all the steps that have to be performed. It creates the physical volume and creates the volume group using the physical volume. Then, it creates the logical volume and creates a filesystem on that logical volume.

To show the `lvm` module in action, we'll create an `lvm` node that has a boot device and a second drive. On my system, the first device is `/dev/sda` and the second drive is `/dev/sdb`. We can see the disk layout using `lsblk` as shown in the following screenshot:

```
[thomas@lvm ~]$ lsblk
NAME      MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda        8:0    0   15G  0 disk
└─sda1     8:1    0  500M  0 part /boot
  └─sda2     8:2    0 14.5G  0 part
    ├─sl_sdl71-swap 253:0    0  1.5G  0 lvm  [SWAP]
```

We can see that `/dev/sdb` is available on the system but nothing is installed on it. We'll create a new module called `lvm_web`, which will create a logical volume of 4 GB, and format it with an `ext4` filesystem, as shown in the following code:

```
class lvm_web {
  lvm::volume {"lv_var_www":
    ensure => present,
    vg     => "vg_web",
    pv     => "/dev/sdb",
    fstype => "ext4",
    size   => "4G",
  }
}
```

Now we'll create an `lvm.yaml` file in `hieradata/hosts/lvm.yaml`:

```
---
welcome: 'lvm node'
classes:
  - lvm_web
```

Now when we run Puppet agent on `lvm`, we will see that the `vg_web` volume group is created, followed by the `lv_var_www` logical volume, and the filesystem after that:

```
Notice: /Stage[main]/Lvm_web/Lvm::Volume[lv_var_www]/Physical_volume[/
/dev/sdb]/ensure: created
Notice: /Stage[main]/Lvm_web/Lvm::Volume[lv_var_www]/Volume_group[vg_
web]/ensure: created
Notice: /Stage[main]/Lvm_web/Lvm::Volume[lv_var_www]/Logical_volume[lv_
var_www]/ensure: created
Notice: /Stage[main]/Lvm_web/Lvm::Volume[lv_var_www]/Filesystem[/dev/vg_
web/lv_var_www]/ensure: created
```

Now when we run `lsblk` again, we will see that the filesystem was created:

```
[thomas@lvm ~]$ lsblk
NAME           MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda            8:0    0   15G  0 disk 
└─sda1          8:1    0  500M  0 part /boot
  └─sda2          8:2    0 14.5G  0 part
    ├─sl_sdl71-swap 253:0    0   1.5G  0 lvm   [SWAP]
    ├─sl_sdl71-root 253:1    0   13G  0 lvm   /
sdb            8:16   0    8G  0 disk
```

Note that the filesystem is not mounted yet, only created. To make this a fully functional class, we would need to add the mount location for the filesystem and ensure that the mount point exists, as shown in the following code:

```
file {'/var/www/html':
  ensure => 'directory',
  owner  => '48',
  group  => '48',
  mode   => '0755',
}

mount {'lvm_web_var_www':
  name  => '/var/www/html',
  ensure => 'mounted',
  device  => "/dev/vg_web/lv_var_www",
  dump   => '1',
  fstype  => "ext4",
  options => "defaults",
  pass    => '2',
  target  => '/etc/fstab',
  require => [Lvm::Volume["lv_var_www"],File["/var/www/html"]],
}
```

Now when we run Puppet again, we can see that the directories are created and the filesystem is mounted:

```
[root@lvm ~]# puppet agent -t
...
Info: Applying configuration version '1443524661'
Notice: /Stage[main]/Lvm_web/File[/var/www/html]/ensure: created
Notice: /Stage[main]/Lvm_web/Mount[lvm_web_var_www]/ensure: defined
'ensure' as 'mounted'
Info: /Stage[main]/Lvm_web/Mount[lvm_web_var_www]: Scheduling refresh of
Mount[lvm_web_var_www]
```

```
Info: Mount[lvm_web_var_www] (provider=parsed): Remounting
Notice: /Stage[main]/Lvm_web/Mount[lvm_web_var_www]: Triggered 'refresh'
from 1 events
Info: /Stage[main]/Lvm_web/Mount[lvm_web_var_www]: Scheduling refresh of
Mount[lvm_web_var_www]
Notice: Finished catalog run in 1.53 seconds
```

Now when we run `lsblk`, we see the filesystem is mounted, as shown in the following screenshot:

NAME	MAJ:MIN	RM	SIZE	R0	TYPE	MOUNTPOINT
sda	8:0	0	15G	0	disk	
└─sda1	8:1	0	500M	0	part	/boot
└─sda2	8:2	0	14.5G	0	part	
└─sl_sdl71-swap	253:0	0	1.5G	0	lvm	[SWAP]
└─sl_sdl71-root	253:1	0	13G	0	lvm	/
sdb	8:16	0	8G	0	disk	

This module can save you a lot of time. The steps required to set up a new volume group, add a logical volume, format the filesystem correctly, and then mount the filesystem can all be reduced to including a single class on a node.

Standard library

The **standard library** (`stdlib`) is a collection of useful facts, functions, types, and providers not included with the base language. Even if you do not use the items within `stdlib` directly, reading about how they are defined is useful to figure out how to write your own modules.

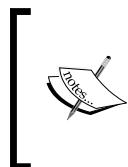
Several functions are provided by `stdlib`; these can be found at <https://forge.puppetlabs.com/puppetlabs/stdlib>. Also, several string-handling functions are provided by it, such as `capitalize`, `chomp`, and `strip`. There are functions for array manipulation and some arithmetic operations such as absolute value (`abs`) and minimum (`min`). When you start building complex modules, the functions provided by `stdlib` can occasionally reduce your code complexity.

Many parts of `stdlib` have been merged into Facter and Puppet. One useful capability originally provided by `stdlib` is the ability to define custom facts based on text files or scripts on the node. This allows processes that run on nodes to supply facts to Puppet to alter the behavior of the agent. To enable this feature, we have to create a directory called `/etc/facter/facts.d` (Puppet enterprise uses `/etc/puppetlabs/facter/facts.d`), as shown here:

```
[root@client ~]# facter -p myfact

[root@client ~]# mkdir -p /etc/facter/facts.d
[root@client ~]# echo "myfact=myvalue" >/etc/facter/facts.d/myfact.txt
[root@client ~]# facter -p myfact
myvalue
```

The `facter_dot_d` mechanism can use text files, YAML, or JSON files based on the extension, `.txt`, `.yaml` or `.json`. If you create an executable file, then it will be executed and the results parsed for fact values as though you had a `.txt` file (`fact = value`).



If you are using a Facter version earlier than 1.7, then you will need the `facter.d` mechanism provided by `stdlib`. This was removed in `stdlib` version 3 and higher; the latest stable `stdlib` version that provides `facter.d` is 2.6.0. You will also need to enable `pluginsync` on your nodes (the default setting on Puppet 2.7 and higher).

To illustrate the usefulness, we will create a fact that returns the gems installed on the system. I'll run this on a host with a few gems to illustrate the point. Place the following script in `/etc/facter/facts.d/gems.sh` and make it executable (`chmod +x gems.sh`):

```
#!/bin/bash

gems=$( /usr/bin/gem list --no-versions | /bin/grep -v "^\$" | /usr/bin/
paste -sd ",")
echo "gems=$gems"
```

Now make the script executable (`chmod 755 /etc/facter/facts.d/gem.sh`) and run Facter to see the output from the fact:

```
[root@client ~]# facter -p gems
bigdecimal,commander,highline,io-console,json,json_pure,psych,puppet-
lint,rdoc
```

We can now use these gems fact in our manifests to ensure that the gems we require are available. Another use of this fact mechanism could be to obtain the version of an installed application that doesn't use normal package-management methods. We can create a script that queries the application for its installed version and returns this as a fact. We will cover this in more detail when we build our own custom facts in a later chapter.

Summary

In this chapter, we have explored how to pull in modules from Puppet Forge and other locations. We looked at methods for keeping our public modules in order such as `librarian-puppet` and `r10k`. We revised our Git hooks to use `r10k` and created an automatic system to update public modules. We then examined a selection of the Forge modules that are useful in the enterprise.

In the next chapter, we will start writing our own custom modules.

5

Custom Facts and Modules

We created and used modules up to this point when we installed and configured tuned using the `is_virtual` fact. We created a module called `virtual` in the process. Modules are nothing more than organizational tools, manifests, and plugin files that are grouped together.

We mentioned `pluginsync` in the previous chapter. By default, in Puppet 3.0 and higher, plugins in modules are synchronized from the master to the nodes. Plugins are special directories in modules that contain Ruby code.

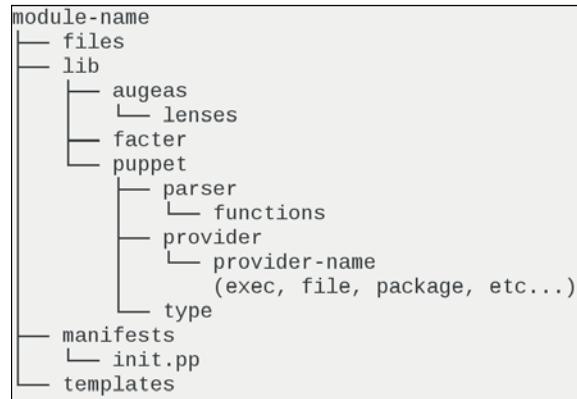
Plugins are contained within the `/lib` subdirectory of a module, and there can be four possible subdirectories defined: `files`, `manifests`, `templates`, and `lib`. The `manifests` directory holds our manifests, as we know `files` has our files, `templates` has the templates, and `lib` is where we extend Augeas, Hiera, Facter, and/or Puppet depending on the files we place there.



You may also see a `spec` directory in modules downloaded from Puppet Forge. This directory holds the files used in testing Puppet code.

In this chapter, we will cover how to use the `modulename/lib/facter` directory to create custom facts, and in subsequent chapters, we will see how to use the `/lib/puppet` directory to create custom types.

The structure of a module is shown in the following diagram:



A module is a directory within the `modulepath` setting of Puppet, which is searched when a module is included by name in a node manifest. If the module name is `base` and our `modulepath` is `$codedir/environments/$environment/modules:$codedir/environments/$environment/dist:$codedir/environments/production/modules`, then the search is done as follows (assuming `codedir` is `/etc/puppetlabs/code`):

```
/etc/puppetlabs/code/environments/$environment/modules/base/manifests/  
init.pp  
/etc/puppetlabs/code/environments/$environment/modules/dist/base/  
manifests/init.pp  
/etc/puppetlabs/code/environments/production/modules/base/manifests/  
init.pp
```

Module manifest files

Each module is expected to have an `init.pp` file defined, which has the top-level class definition; in the case of our `base` example, `init.pp` is expected to contain `class base { }`.

Now, if we include `base::subitem` in our node manifest, then the file that Puppet will search for will be `base/manifests/subitem.pp`, and that file should contain `class base::subitem { }`.

It is also possible to have subdirectories of the `manifests` directory defined to split up the manifests even more. As a rule, a manifest within a module should only contain a single class. If we wish to define `base::subitem::subsetting`, then the file will be `base/manifests/subitem/subsetting.pp`, and it would contain `class base::subitem::subsetting { }`.

Naming your files correctly means that they will be loaded automatically when needed, and you won't have to use the `import` function (the `import` function is deprecated in version 3 and completely removed in version 4). By creating multiple subclasses, it becomes easy to separate a module into its various components; this is important later when you need to include only parts of the module in another module. As an example, say we have a database system called `judy`, and `judy` requires the `judy-server` package to run. The `judy` service requires the users `judy` and `judyadm` to run. Users `judy` and `judyadm` require the `judygrp` group, and they all require a filesystem to contain the database. We will split up these various tasks into separate manifests. We'll sketch the contents of this fictional module, as follows:

- In `judy/manifests/groups.pp`, we'll have the following code:

```
class judy::groups {
    group {'judygrp': }

}
```

- In `judy/manifests/users.pp`, we'll have the following code:

```
class judy::users {
    include judy::groups
    user {'judy':
        require => Group['judygrp']
    }
    user {'judyadm':
        require => Group['judygrp']
    }
}
```

- In `judy/manifests/packages.pp`, we'll have the following code:

```
class judy::packages {
    package {'judy-server':
        require => User['judy', 'judyadm']
    }
}
```

- In `judy/manifests/filesystem.pp`, we'll have the following code:

```
class judy::filesystem {
    lvm {'/opt/judy':
        require => File['/opt/judy']
    }
    file {'/opt/judy': }
}
```

- Finally, our service starts from `judy/manifests/service.pp`:

```
class judy::service {
    service {'judy':
        require => [
            Package['judy-server'],
            File['/opt/judy'],
            Lvm['/opt/judy'],
            User['judy', 'judyadm']
        ],
    }
}
```

Now, we can include each one of these components separately, and our node can contain `judy::packages` or `judy::service` without using the entire `judy` module. We will define our top level module (`init.pp`) to include all these components, as shown here:

```
class judy {
    include judy::users
    include judy::group
    include judy::packages
    include judy::filesystem
    include judy::service
}
```

Thus, a node that uses `include judy` will receive all of those classes, but if we have a node that only needs the `judy` and `judyadm` users, then we need to include only `judy::users` in the code.

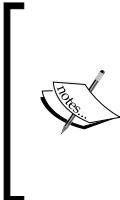
Module files and templates

Transferring files with Puppet is something that is best done within modules. When you define a file resource, you can either use `content => "something"` or you can push a file from the Puppet master using `source`. For example, using our `judy` database, we can have `judy::config` with the following file definition:

```
class judy::config {
  file {'/etc/judy/judy.conf':
    source => 'puppet:///modules/judy/judy.conf'
  }
}
```

Now, Puppet will search for this file in the `[modulepath]/judy/files` directory. It is also possible to add full paths and have your module mimic the filesystem. Hence, the previous source line will be changed to `source => 'puppet:///modules/judy/etc/judy/judy.conf'`, and the file will be found at `[modulepath]/judy/files/etc/judy/judy.conf`.

The `puppet:///` URI source line mentioned earlier has three backslashes; optionally, the name of a puppetserver may appear between the second and third backslash. If this field is left blank, the puppetserver that performs the catalog compilation will be used to retrieve the file. You can alternatively specify the server using `source => 'puppet://puppetfile.example.com/modules/judy/judy.conf'`.

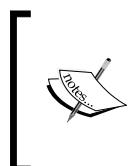


Having files that come from specific puppetservers can make maintenance difficult. If you change the name of your puppetserver, you have to change all references to that name as well. Puppet is not ideal for transferring large files, if you need to move large files onto your machines, consider using the native packaging system of your client nodes.

Templates are searched in a similar fashion. In this example, to specify the template in `judy/templates`, you will use `content =>template('judy/template.erb')` to have Puppet look for the template in your modules' `templates` directory. For example, another config file for `judy` can be defined, as follows:

```
file {'/etc/judy/judyadm.conf':
  content => template('judy/judyadm.conf.erb')
}
```

Puppet will look for the '`judy/judyadm.conf.erb`' file at `[modulepath]/judy/templates/judyadm.conf.erb`. We haven't covered the **Embedded Ruby (ERB)** templates up to this point; templates are files that are parsed according to the ERB syntax rules. If you need to distribute a file where you need to change some settings based on variables, then a template can help. The ERB syntax is covered in detail at <http://docs.puppetlabs.com/guides/template.html>. Puppet 4 (and Puppet 3 with the future parser enabled) supports EPP templates as well. EPP templates are Embedded Puppet templates that use Puppet language syntax rather than Ruby.



ERB templates were used by many people to overcome the inability to perform iteration with Puppet. EPP is the newer templating engine that doesn't rely on Ruby. EPP is the currently recommended templating engine. If you are starting from scratch, I would recommend using EPP syntax templates.



Modules can also include custom facts, as we've already seen in this chapter. Using the `lib` subdirectory, it is possible to modify both Facter and Puppet. In the next section, we will discuss module implementations in a large organization before writing custom modules.

Naming a module

Modules must begin with a lowercase letter and only contain lowercase letters, numbers, and the underscore (`_`) symbol. No other characters should be used. While writing modules that will be shared across the organization, use names that are obvious and won't interfere with other groups' modules or modules from the Forge. A good rule of thumb is to insert your corporation's name at the beginning of the module name and, possibly, your group name.



While uploading to the Forge, your Forge username will be prepended to the module (`username-modulename`).



While designing modules, each module should have a specific purpose and not pull in manifests from other modules and each one of them should be autonomous. Classes should be used within the module to organize functionality. For instance, a module named `example_foo` installs a package and configures a service. Now, separating these two functions and their supporting resources into two classes, `example_foo::pkg` and `example_foo::svc`, will make it easier to find the code you need to work on, when you need to modify these different components. In addition, when you have all the service accounts and groups in another file, it makes it easier to find them, as well.

Creating modules with a Puppet module

To start with a simple example, we will use Puppet's `module` command to generate empty module files with comments. The module name will be `example_phpmyadmin`, and the `generate` command expects the generated argument to be `[our username] - [module name]`; thus, using our sample developer, `samdev`, the argument will be `samdev-example_phpmyadmin`, as shown here:

```
[samdev@stand ~]$ cd control/dist/  
[samdev@standdist]$ puppet module generate samdev-example_phpmyadmin  
We need to create a metadata.json file for this module. Please answer  
the  
following questions; if the question is not applicable to this module,  
feel free  
to leave it blank.  
  
Puppet uses Semantic Versioning (semver.org) to version modules.  
What version is this module? [0.1.0]  
--> 0.0.1  
  
Who wrote this module? [samdev]  
-->  
  
What license does this module code fall under? [Apache-2.0]  
-->  
  
How would you describe this module in a single sentence?  
--> An Example Module to install PHPMyAdmin  
  
Where is this module's source code repository?  
--> https://github.com/uphillian  
  
Where can others go to learn more about this module? [https://github.  
com/uphillian]  
-->  
  
Where can others go to file issues about this module? [https://github.  
com/uphillian/issues]  
-->
```

```
-----  
{  
  "name": "samdev-example_phpmyadmin",  
  "version": "0.0.1",  
  "author": "samdev",  
  "summary": "An Example Module to install PHPMyAdmin",  
  "license": "Apache-2.0",  
  "source": "https://github.com/uphillian",  
  "project_page": "https://github.com/uphillian",  
  "issues_url": "https://github.com/uphillian/issues",  
  "dependencies": [  
    {"name": "puppetlabs-stdlib", "version_requirement": ">= 1.0.0"}  
  ]  
}  
-----
```

```
About to generate this metadata; continue? [n/Y]  
-->y
```

```
Notice: Generating module at /home/samdev/control/dist/example_  
phpmyadmin...  
Notice: Populating templates...  
Finished; module generated in example_phpmyadmin.  
example_phpmyadmin/manifests  
example_phpmyadmin/manifests/init.pp  
example_phpmyadmin/spec  
example_phpmyadmin/spec/classes  
example_phpmyadmin/spec/classes/init_spec.rb  
example_phpmyadmin/spec/spec_helper.rb  
example_phpmyadmin/tests  
example_phpmyadmin/tests/init.pp  
example_phpmyadmin/Gemfile  
example_phpmyadmin/Rakefile  
example_phpmyadmin/README.md  
example_phpmyadmin/metadata.json
```

 If you plan to upload your module to the Forge or GitHub, use your Forge or GitHub account name for the user portion of the module name (in the example, replace `samdev` with your GitHub account).

Comments in modules

The previous command generates `metadata.json` and `README.md` files that can be modified for your use as and when required. The `metadata.json` file is where you specify who wrote the module and which license it is released under. If your module depends on any other module, you can specify the modules in the `dependencies` section of this file. In addition to the `README.md` file, an `init.pp` template is created in the `manifests` directory.

Our `phpmyadmin` package needs to install Apache (`httpd`) and configure the `httpd` service, so we'll create two new files in the `manifests` directory, `pkg.pp` and `svc.pp`.

 It's important to be consistent from the beginning; if you choose to use `package.pp` and `service.pp`, use that everywhere to save yourself time later.

In `init.pp`, we'll include our `example_phpmyadmin::pkg` and `example_phpmyadmin::svc` classes, as shown in the following code:

```
class example_phpmyadmin {
    include example_phpmyadmin::pkg
    include example_phpmyadmin::svc
}
```

The `pkg.pp` file will define `example_phpmyadmin::pkg`, as shown in the following code:

```
class example_phpmyadmin::pkg {
    package {'httpd':
        ensure => 'installed',
        alias  => 'apache'
    }
}
```

The `svc.pp` file will define `example_phpmyadmin::svc`, as shown in the following code:

```
class example_phpmyadmin::svc {
    service {'httpd':
        ensure => 'running',
        enable => true
    }
}
```

Now, we'll define another module called `example_phpldapadmin` using the `puppet module generate` command, as shown here:

```
[samdev@standdist] $ puppet module generate samdev-example_phpldapadmin
We need to create a metadata.json file for this module. Please answer
the
following questions; if the question is not applicable to this module,
feel free
to leave it blank.

...
Notice: Generating module at /home/samdev/control/dist/example_
phpldapadmin...
Notice: Populating templates...
Finished; module generated in example_phpldapadmin.
example_phpldapadmin/manifests
example_phpldapadmin/manifests/init.pp
example_phpldapadmin/spec
example_phpldapadmin/spec/classes
example_phpldapadmin/spec/classes/init_spec.rb
example_phpldapadmin/spec/spec_helper.rb
example_phpldapadmin/tests
example_phpldapadmin/tests/init.pp
example_phpldapadmin/Gemfile
example_phpldapadmin/Rakefile
example_phpldapadmin/README.md
example_phpldapadmin/metadata.json
```

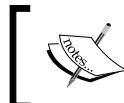
We'll define the `init.pp`, `pkg.pp`, and `svc.pp` files in this new module just as we did in our last module so that our three class files contain the following code:

```
class example_phpldapadmin {
    include example_phpldapadmin::pkg
    include example_phpldapadmin::svc
}

class example_phpldapadmin::pkg {
    package {'httpd':
        ensure => 'installed',
        alias  => 'apache'
    }
}

class example_phpldapadmin::svc {
    service {'httpd':
        ensure => 'running',
        enable  => true
    }
}
```

Now we have a problem, `phpldapadmin` uses the `httpd` package and so does `phpmyadmin`, and it's quite likely that these two modules may be included in the same node.



Remember to add the two modules to your control repository and push the changes to Git. Your Git hook should trigger an update to Puppet module directories.

We'll include both of them on our client by editing `client.yaml` and then we will run Puppet using the following command:

```
[root@client ~]# puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts

Error: Could not retrieve catalog from remote server: Error 400 on
 SERVER: Evaluation Error: Error while evaluating a Resource Statement,
 Duplicate declaration: Package[httpd] is already declared in file /
 /etc/puppetlabs/code/environments/production/dist/example_phpmyadmin/
 manifests/pkg.pp:2; cannot redeclare at /etc/puppetlabs/code/
 environments/production/dist/example_phpldapadmin/manifests/pkg.pp:2 at
 /etc/puppetlabs/code/environments/production/dist/example_phpldapadmin/
```

```
manifests/pkg.pp:2:3 on node client.example.com
Warning: Not using cache on failed catalog
Error: Could not retrieve catalog; skipping run
```

Multiple definitions

A resource in Puppet can only be defined once per node. What this means is that if our module defines the `httpd` package, no other module can define `httpd`. There are several ways to deal with this problem and we will work through two different solutions.

The first solution is the more difficult option—use **virtual resources** to define the package and then realize the package in each place you need. Virtual resources are similar to a placeholder for a resource; you define the resource but you don't use it. This means that Puppet master knows about the Puppet definition when you *virtualize* it, but it doesn't include the resource in the catalog at that point. Resources are included when you realize them later; the idea being that you can virtualize the resources multiple times and not have them interfere with each other. Working through our example, we will use the @ (at) symbol to virtualize our package and service resources. To use this model, it's helpful to create a container for the resources you are going to virtualize. In this case, we'll make modules for `example_packages` and `example_services` using Puppet module's `generate` command again.

The `init.pp` file for `example_packages` will contain the following:

```
class example_packages {
  @package {'httpd':
    ensure => 'installed',
    alias  => 'apache',
  }
}
```

The `init.pp` file for `example_services` will contain the following:

```
class example_services {
  @service {'httpd':
    ensure  => 'running',
    enable  => true,
    require => Package['httpd'],
  }
}
```

These two classes define the package and service for `httpd` as virtual. We then need to include these classes in our `example_phpmyadmin` and `example_phpldapadmin` classes. The modified `example_phpmyadmin::pkg` class will now be, as follows:

```
class example_phpmyadmin::pkg {
  include example_packages
  realize(Package['httpd'])
}
```

And the `example_phpmyadmin::svc` class will now be the following:

```
class example_phpmyadmin::svc {
  include example_services
  realize(Service['httpd'])
}
```

We will modify the `example_phpldapadmin` class in the same way and then attempt another Puppet run on client (which still has `example_phpldapadmin` and `example_phpmyadmin` classes), as shown here:

```
[root@client ~]# puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for client.example.com
Info: Applying configuration version '1443928369'
Notice: /Stage[main]/Example_packages/Package[httpd]/ensure: created
Notice: /Stage[main]/Example_services/Service[httpd]/ensure: ensure
changed 'stopped' to 'running'
Info: /Stage[main]/Example_services/Service[httpd]: Unscheduling refresh
on Service[httpd]
Notice: Applied catalog in 11.17 seconds
```

For this solution to work, you need to migrate the resources that may be used by multiple modules to your top-level resource module and include the resource module wherever you need to realize the resource.

In addition to the `realize` function, used previously, a collector exists for virtual resources. A **collector** is a kind of glob that can be applied to virtual resources to realize resources based on a tag. A tag in Puppet is just a meta attribute of a resource that can be used for searching later. Tags are only used by collectors (for both virtual and exported resources, the exported resources will be explored in a later chapter) and they do not affect the resource.

To use a collector in the previous example, we will have to define a tag in the virtual resources, for the `httpd` package this will be, as follows:

```
class example_packages {
  @package {'httpd':
    ensure => 'installed',
    alias  => 'apache',
    tag    => 'apache',
  }
}
```

And then to realize the package using the collector, we will use the following code:

```
class example_phpldapadmin::pkg {
  include example_packages
  Package <| tag == 'apache' |>
}
```

The second solution will be to move the resource definitions into their own class and include that class whenever you need to realize the resource. This is considered to be a more appropriate way of solving the problem. Using the virtual resources described previously splits the definition of the package away from its use area.

For the previous example, instead of a class for all package resources, we will create one specifically for Apache and include that wherever we need to use Apache. We'll create the `example_apache` module monolithically with a single class for the package and the service, as shown in the following code:

```
class example_apache {
  package {'httpd':
    ensure => 'installed',
    alias  => 'apache'
  }
  service {'httpd':
    ensure  => 'running',
    enable  => true,
    require=> Package['httpd'],
  }
}
```

Now, in `example_phpldapadmin::pkg` and `example_phpldapadmin::svc`, we only need to include `example_apache`. This is because we can include a class any number of times in a catalog compilation without error. So, both our `example_phpldapadmin::pkg` and `example_phpldapadmin::svc` classes are going to receive definitions for the package and service of `httpd`; however, this doesn't matter, as they only get included once in the catalog, as shown in the following code:

```
class example_phpldapadmin::pkg {
    include example_apache
}
```

Both these methods solve the issue of using a resource in multiple packages. The rule is that a resource can only be defined once per catalog, but you should think of that rule as once per organization so that your modules won't interfere with those of another group within your organization.

Custom facts

While managing a complex environment, facts can be used to bring order out of chaos. If your manifests have large `case` statements or nested `if` statements, a custom fact might help in reducing the complexity or allow you to change your logic.

When you work in a large organization, keeping the number of facts to a minimum is important, as several groups may be working on the same system and thus interaction between the users may adversely affect one another's work or they may find it difficult to understand how everything fits together.

As we have already seen in the previous chapter, if our facts are simple text values that are node specific, we can just use the `facts.d` directory of `stdlib` to create static facts that are node specific.

This `facts.d` mechanism is included, by default, on Facter versions 1.7 and higher and is referred to as external fact.

Creating custom facts

We will be creating some custom facts; therefore, we will create our Ruby files in the `module_name/lib/facter` directory. While designing your facts, choose names that are specific to your organization. Unless you plan on releasing your modules on the Forge, avoid calling your fact something similar to a predefined fact or using a name that another developer might use. The names should be meaningful and specific—a fact named `foo` is probably not a good idea. Facts should be placed in the specific module that requires them. Keeping the fact name related to the module name will make it easier to determine where the fact is being set later.

For our `example.com` organization, we'll create a module named `example_facts` and place our first fact in there. As the first example, we'll create a fact that returns `1` (true) if the node is running the latest installed kernel or `0` (false) if not. As we don't expect this fact to become widely adopted, we'll call it `example_latestkernel`. The idea here is that we can apply modules to nodes that are not running the latest installed kernel, such as locking them down or logging them more closely.

To begin writing the fact, we'll start writing a Ruby script; you can also work in IRB while you're developing your fact. **Interactive Ruby (IRB)** is like a shell to write the Ruby code, where you can test your code instantly. Version 4 of Puppet installs its own Ruby, so our fact will need to use the Ruby installed by Puppet (`/opt/puppetlabs/puppet/bin/ruby`). Our fact will use a function from Puppet, so we will require `puppet` and `facter`. The fact scripts are run from within Facter so that the `require` lines are removed once we are done with our development work. The script is written, as follows:

```
#!/opt/puppetlabs/puppet/bin/ruby
require 'puppet'
require 'facter'
# drop alpha numeric endings
def sanitize_version(version)
  temp = version.gsub(/.(el5|el6|el7|fc19|fc20)/, '')
  return temp.gsub(/.(x86_64|i686|i586|i386)/, '')
end
```

We define a function to remove textual endings on kernel versions and architectures. Textual endings, such as `el5` and `el6` will make our version comparison return incorrect results. For example, `2.6.32-431.3.1.el6` is less than `2.6.32-431.el6` because the `e` in `el6` is higher in ASCII than `3`. Our script will get simplified greatly, if we simply remove known endings. We then obtain a list of installed kernel packages; the easiest way to do so is with `rpm`, as shown here:

```
kernels = %x( rpm -q kernel --qf '%{version}-%{release}\n' )
kernels = sanitize_version(kernels)
latest = ''
```

We will then set the `latest` variable to empty and we'll loop through the installed kernels by comparing them to `latest`. If their values are greater than `latest`, then we convert `latest` such that it is equal to the value of the kernels. At the end of the loop, we will have the `latest` (largest version number) kernel in the variable. For `kernel` in `kernels`, we will use the following commands:

```
for kernel in kernels.split('\n')
  kernel=kernel.chomp()
  if latest == ''
```

```
    latest = kernel
  end
  if Puppet::Util::Package.versioncmp(kernel,latest) > 0
    latest = kernel
  end
end
```

We use `versioncmp` from `puppet::util::package` to compare the versions. I've included a debugging statement in the following code that we will remove later. At the end of this loop, the `latest` variable contains the largest version number and the latest installed kernel:

```
kernelrelease = Facter.value('kernelrelease')
kernelrelease = sanitize_version(kernelrelease)
```

Now, we will ask Facter for the value of `kernelrelease`. We don't need to run `uname` or a similar tool, as we'll rely on Facter to get the value using the `Facter.value('kernelrelease')` command. Here, `Facter.value()` returns the value of a known fact. We will also run the result of `Facter.value()` through our `sanitize_version` function to remove textual endings. We will then compare the value of `kernelrelease` with `latest` and update the `kernellatest` variable accordingly:

```
if Puppet::Util::Package.versioncmp(kernelrelease,latest) == 0
  kernellatest = 1
else
  kernellatest = 0
end
```

At this point, `kernellatest` will contain the value `1` if the system is running the installed kernel with `latest` and `0` if not. We will then print some debugging information to confirm whether our script is doing the right thing, as shown here:

```
print "running kernel = %s\n" % kernelrelease
print "latest installed kernel = %s\n" % latest
print "kernellatest = %s\n" % kernellatest
```

We'll now run the script on `node1` and compare the results with the output of `rpm -q kernel` to check whether our fact is calculating the correct value:

```
[samdev@standfacter] $ rpm -q kernel
kernel-3.10.0-229.11.1.el7.x86_64
kernel-3.10.0-229.14.1.el7.x86_64
[samdev@standfacter] $ ./latestkernel.rb
3.10.0-229.11.1.el7
```

```
3.10.0-229.14.1.el7
running kernel = 3.10.0-229.11.1.el7
latest installed kernel = 3.10.0-229.11.1.el7
3.10.0-229.14.1.el7
kernellatest = 0
```

Now that we've verified that our fact is doing the right thing, we need to call `Facter.add()` to add a fact to Facter. The reason behind this will become clear in a moment, but we will place all our code within the `Facter.add` section, as shown in the following code:

```
Facter.add("example_latestkernel") do
  kernels = %x( rpm -q kernel --qf '%{version}-%{release}\n' )
  ...
end
Facter.add("example_latestkernelinstalled") do
  setcode do latest end
end
```

This will add two new facts to Facter. We now need to go back and remove our require lines and print statements. The complete fact should look similar to the following script:

```
# drop alpha numeric endings
def sanitize_version (version)
  temp = version.gsub(/.(el5|el6|el7|fc19|fc20)/,'')
  return temp.gsub(/.(x86_64|i686|i586|i386)/,'')
end
Facter.add("example_latestkernel") do
  kernels = %x( rpm -q kernel --qf '%{version}-%{release}\n' )
  kernels = sanitize_version(kernels)
  latest = ''
  for kernel in kernels do
    kernel=kernel.chomp()
    if latest == ''
      latest = kernel
    end
    if Puppet::Util::Package.versioncmp(kernel,latest) > 0
      latest = kernel
    end
  end
  kernelrelease = Facter.value('kernelrelease')
  kernelrelease = sanitize_version(kernelrelease)
```

```
if Puppet::Util::Package.versioncmp(kernelrelease,latest) == 0
    kernellatest = 1
else
    kernellatest = 0
end
setcode do kernellatest end
Facter.add("example_latestkernelinstalled") do
    setcode do latest end
end
end
```

Now, we need to create a module of our Git repository on stand and have that checked out by client to see the fact in action. Switch back to the samdev account to add the fact to Git as follows:

```
[Thomas@stand ~]$ sudo -iu samdev
[samdev@stand]$ cd control/dist
[samdev@stand]$ mkdir -p example_facts/lib/facter
[samdev@stand]$ cd example_facts/lib/facter
[samdev@stand]$ cp ~/latestkernel.rb example_latestkernel.rb
[samdev@stand]$ git add example_latestkernel.rb
[samdev@stand]$ git commit -m "adding first fact to example_facts"
[masterd42bc22] adding first fact to example_facts
 1 files changed, 33 insertions(+), 0 deletions(-)
create mode 100755 dist/example_facts/lib/facter/example_latestkernel.rb
[samdev@stand]$ git push origin
...
To /var/lib/git/control.git/
fc4f2e5..55305d8 production -> production
```

Now, we will go back to client, run Puppet agent, and see that example_latestkernel.rb is placed in /opt/puppetlabs/puppet/cache/lib/facter/example_latestkernel.rb so that Facter can now use the new fact.

This fact will be in the /dist folder of the environment. In the previous chapter, we added /etc/puppet/environments/\$environment/dist to modulepath in puppet.conf; if you haven't done this already, do so now:

```
[root@client ~]# puppet agent -t
...
Notice: /File[/opt/puppetlabs/puppet/cache/lib/facter/example_
latestkernel.rb]/ensure: defined content as '{md5}579a2f06068d4a9f40d1dad
```

```
cd2159527'...
Notice: Finished catalog run in 1.18 seconds
[root@client ~]# facter -p |grep ^example
example_latestkernel => 1
example_latestkernelinstalled => 3.10.0-123
```

Now, this fact works fine for systems that use rpm for package management; it will not work on an apt system. To ensure that our fact doesn't fail on these systems, we can use a `confine` statement to confine the fact calculation to systems where it will succeed. We can assume that our script will work on all systems that report RedHat for the `osfamily` fact, so we will confine ourselves to that fact.

For instance, if we run Puppet on a Debian-based node to apply our custom fact, it fails when we run Facter, as shown here:

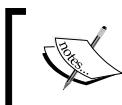
```
# cat /etc/debian_version
wheezy/sid
# facter -p example_latestkernelinstalled
sh: 1: rpm: not found
Could not retrieve example_latestkernelinstalled: undefined local
variable or method `latest' for #<Facter::Util::Resolution:0xb6bd386c>
```

Now, if we add a `confine` statement to confine the fact to nodes in which `osfamily` is RedHat, it doesn't happen, as shown here:

```
Facter.add("example_latestkernel") do
  confine :osfamily => 'RedHat'
  ...
end
Facter.add("example_latestkernelinstalled") do
  confine :osfamily => 'RedHat'
  setcode do latest end
end
```

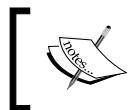
When we run Facter on the Debian node again, we will see that the fact is simply not defined, as shown here:

```
# facter -p example_latestkernelinstalled
##
```



In the previous command, the prompt is returned without an error, and the `confine` statements prevent the fact from being defined, so there is no error to return.

This simple example creates two facts that can be used in modules. Based on this fact you can, for instance, add a warning to `motd` to say that the node needs to reboot.

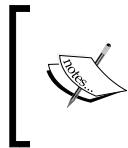


If you want to become really popular at work, have the node turn off SSH until it's running the latest kernel in the name of security.



While implementing a custom fact such as this, every effort should be made to ensure that the fact doesn't break Facter compilation on any OSes within your organization. Using `confine` statements is one way to ensure your facts stay where you designed them.

So, why not just use the external fact (`/etc/facter/facts.d`) mechanism all the time? We could have easily written the previous fact script in bash and put the executable script in `/etc/facter/facts.d`. Indeed, there is no problem in doing it that way. The problem with using the external fact mechanism is timing and precedence. The fact files placed in `lib/facter` are synced to nodes when `pluginsync` is set to `true`, so the custom fact is available for use during the initial catalog compilation. If you use the external fact mechanism, you have to send your script or text file to the node during the agent run so that the fact isn't available until after the file has been placed there (after the first run, any logic built around that fact will be broken until the next Puppet run). The second problem is preference. External facts are given a very high weight by default. Weight in the Facter world is used to determine when a fact is calculated and facts with low weight are calculated first and cannot be overridden by facts with higher weight.



Weights are often used when a fact can be determined by one of the several methods. The preferred method is given the lowest weight. If the preferred method is unavailable (due to a `confine`), then the next higher weight fact is tried.



One great use case for external facts is having a system task (something that runs out of cron perhaps) that generates the text file in `/etc/facter/facts.d`. Initial runs of Puppet agent won't see the fact until after cron runs the script, so you can use this to trigger further configuration by having your manifests key off the new fact. As a concrete example, you can have your node installed as a web server for a load-balancing cluster as a part of the modules that run a script from cron to ensure that your web server is up and functioning and ready to take a part of the load. The cron script will then define a `load_balancer_ready=true` fact. It will then be possible to have the next Puppet agent run and add the node to the load balancer configuration.

Creating a custom fact for use in Hiera

The most useful custom facts are those that return a calculated value that you can use to organize your nodes. Such facts allow you to group your nodes into smaller groups or create groups with similar functionality or locality. These facts allow you to separate the data component of your modules from the logic or code components. This is a common theme that will be addressed again in *Chapter 9, Roles and Profiles*. This can be used in your `hiera.yaml` file to add a level to the hierarchy. One aspect of the system that can be used to determine information about the node is the IP address. Assuming that you do not reuse the IP addresses within your organization, the IP address can be used to determine where or in which part a node resides on a network, specifically, the zone. In this example, we will define three zones in which the machines reside: production, development, and sandbox. The IP addresses in each zone are on different subnets. We'll start by building a script to calculate the zone and then turn it into a fact similar to our last example. Our script will need to calculate IP ranges using netmasks, so we'll import the `ipaddr` library and use the `IPAddr` objects to calculate ranges:

```
require('ipaddr')
require('facter')
require('puppet')
```

Next, we'll define a function that takes an IP address as the argument and returns the zone to which that IP address belongs:

```
def zone(ip)
  zones = {
    'production' => [IPAddr.new('10.0.2.0/24'),IPAddr.
new('192.168.124.0/23')],
    'development' => [IPAddr.new('192.168.123.0/24'),IPAddr.
new('192.168.126.0/23')],
    'sandbox' => [IPAddr.new('192.168.128.0/22')]
  }
  for zone in zones.keys do
    for subnet in zones[zone] do
      if subnet.include?(ip)
        return zone
      end
    end
  end
  return 'undef'
end
```

This function will loop through the zones looking for a match on the IP address. If no match is found, the value of `undef` is returned. We then obtain an IP address for the machine that is using the IP address fact from Facter:

```
ip = IPAddr.new(Facter.value('ipaddress'))
```

Then, we will call the `zone` function with this IP address to obtain the zone:

```
print zone(ip), "\n"
```

Now, we can make this script executable and test it:

```
[root@client ~]# facter ipaddress
10.0.2.15
[root@client ~]# ./example_zone.rb
production
```

Now, all we have to do is replace `print zone(ip), "\n"` with the following code to define the fact:

```
Facter.add('example_zone') do
  setcode do zone(ip) end
end
```

Now, when we insert this code into our `example_facts` module and run Puppet on our nodes, the custom fact is available:

```
[root@client ~]# facter -p example_zone
production
```

Now that we can define a zone based on a custom fact, we can go back to our `hiera.yaml` file and add `%{::example_zone}` to the hierarchy. The `hiera.yaml` hierarchy will now contain the following:

```
---
:hierarchy:
  - "zones/%{::example_zone}"
  - "hosts/%{::hostname}"
  - "roles/%{::role}"
  - "%{::kernel}/%{::osfamily}/%{::lsbmajdistrelease}"
  - "is_virtual/%{::is_virtual}"
  - common
```

After restarting puppetserver to have the `hiera.yaml` file reread, we create a `zones` directory in `hieradata` and add `production.yaml` with the following content:

```
---  
welcome: "example_zone - production"
```

Now when we run Puppet on our `node1`, we will see `motd` updated with the new welcome message, as follows:

```
[root@client ~]# cat /etc/motd  
example_zone - production  
Managed Node: client  
Managed by Puppet version 4.2.2
```

Creating a few key facts that can be used to build up your hierarchy can greatly reduce the complexity of your modules. There are several workflows available, in addition to the custom fact we described earlier. You can use the `/etc/facter/facts.d` (or `/etc/puppetlabs/facter/facts.d`) directory with static files or scripts, or you can have tasks run from other tools that dump files into that directory to create custom facts.

While writing Ruby scripts, you can use any other fact by calling `Facter.value('factname')`. If you write your script in Ruby, you can access any Ruby library using `require`. Your custom fact can query the system using `lspci` or `lsusb` to determine which hardware is specifically installed on that node. As an example, you can use `lspci` to determine the make and model of graphics card on the machine and return that as a fact, such as `videocard`.

CFacter

Facter was earlier written in Ruby and collecting facts about the system through Ruby was a slow process. **CFacter** is a project to rewrite Facter using C++. To enable CFacter in versions of Puppet prior to 4, the `cfacter=true` option will need to be added to `puppet.conf` (this requires Facter version 2.4). As of Facter version 3.0, CFacter is now the default Facter implementation. In my experience, the speedup of Facter is remarkable. On my test system, the Ruby version of Facter takes just under 3 seconds to run. The C++ version of Facter runs in just over 200 milliseconds. Custom Ruby facts are still supported via the Ruby API, as well as facts written in any language via the executable script method.

Summary

In this chapter, we used Ruby to extend Facter and define custom facts. Custom facts can be used in Hiera hierarchies to reduce complexity and organize our nodes. We then began writing our own custom modules and ran into a few problems with multiple defined resources. Two solutions were presented: virtual resources and refactoring the code.

In the next chapter, we will be making our custom modules more useful with custom types.

6

Custom Types

Puppet is about configuration management. As you write more and more code in Puppet, patterns will begin to emerge—sections of code that repeat with minor differences. If you were writing your code in a regular scripting language, you'd reach for a function or subroutine definition at this point. Puppet, similar to other languages, supports the blocking of code in multiple ways; when you reach for functions, you can use defined types; when you overload an operator, you can use a parameterized class, and so on. In this chapter, we will show you how to use parameterized classes and introduce the `define` function to define new user-defined types; following that, we will introduce custom types written in Ruby.

Parameterized classes

Parameterized classes are classes in which you have defined several parameters that can be overridden when you instantiate the class for your node. The use case for parameterized classes is when you have something that won't be repeated within a single node. You cannot define the same parameterized class more than once per node. As a simple example, we'll create a class that installs a database program and starts that database's service. We'll call this class `example::db`; the definition will live in `modules/example/manifests/db.pp`, as follows:

```
class example::db ($db) {
  case $db {
    'mysql': {
      $dbpackage = 'mysql-server'
      $dbservice = 'mysqld'
    }
    'postgresql': {
      $dbpackage = 'postgresql-server'
      $dbservice = 'postgresql'
    }
  }
}
```

```
        }
      package { "$dbpackage": }
      service { "$dbservice":
        ensure  => true,
        enable  => true,
        require  => Package["$dbpackage"]
      }
    }
```

This class takes a single parameter (`$db`) that specifies the type of the database: in this case either `postgresql` or `mysql`. To use this class, we have to instantiate it, as follows:

```
class { 'example::db':
  db => 'mysql'
}
```

Now, when we apply this to a node, we see that `mysql-server` is installed and `mysqld` is started and enabled at boot. This works great for something similar to a database, since we don't think we will have more than one type of database server on a single node. If we try to instantiate the `example::db` class with `postgresql` on our node, we'll get an error, as shown in the following screenshot:



```
Error: Could not retrieve catalog from remote server: Error 400 on SERVER: Evaluation Error: Error while evaluating a Resource Statement: Duplicate declaration: Class[Example::Db] is already declared in file /etc/puppetlabs/code/environments/production/manifests/site.pp:2; cannot redeclare at /etc/puppetlabs/code/environments/production/manifests/site.pp:3 at /etc/puppetlabs/code/environments/production/manifests/site.pp:3:3 on node dbhost
Warning: Not using cache on failed catalog
Error: Could not retrieve catalog; skipping run
```

This fails because we cannot reuse a class on the same node. We'll need to use another structure, the defined type that we'll cover shortly. But first, we'll look at one of the language improvements in Puppet 4.

Data types

The preceding example's parameterized class does not take advantage of the new Puppet language features in version 4. Version 4 of the Puppet language supports explicit data types. Data types in previous versions had to be determined by comparing items and often hoping for the best. This led to some bad practices, such as using the string value `true` to represent the Boolean value `True`. Using the version 4 syntax, we can change the preceding class to require the `$db` parameter to be a string, as shown:

```
class example::db (String $db) {
  case $db {
```

```
'mysql': {
    $dbpackage = 'mysql-server'
    $dbservice = 'mysqld'
}
'postgresql': {
    $dbpackage = 'postgresql-server'
    $dbservice = 'postgresql'
}
}
package { "$dbpackage": }
service { "$dbservice":
    ensure  => true,
    enable  => true,
    require  => Package["$dbpackage"]
}
}
```

The ability to know the type of a parameter has been a long-standing bug with Puppet, particularly when dealing with Boolean values. For more information on the data types supported by Puppet 4, refer to the documentation page at https://docs.puppetlabs.com/puppet/latest/reference/lang_data_type.html.

Defined types

A situation where you have a block of code that is repeated within a single node can be managed with defined types. You can create a defined type with a call to `define`. You can use `define` to refer to a block of Puppet code that receives a set of parameters when instantiated. Our previous database example could be rewritten as a defined type to allow more than one type of database server to be installed on a single node.

Another example of where a defined type is useful is in building filesystems with the LVM module. When we used the LVM module to build a filesystem, there were three things required: we needed a filesystem (a logical volume or LVM resource), a location to mount the filesystem (a file resource), and a mount command (a mount resource). Every time we want to mount a filesystem, we'll need these. To make our code cleaner, we'll create a defined type for a filesystem. Since we don't believe this will be used outside our example organization, we'll call it `example::fs`.

Defined types start with the keyword `define` followed by the name of the defined type and the parameters wrapped in parentheses, as shown in the following code:

```
define example::fs
(
  String $mnt      = "$title",    # where to mount the filesystem
  String $vg       = 'VolGroup',   # which volume group
  String $pv,          # which physical volume
  String $lv,          # which logical volume
  Enum['ext4','ext3','xfs'] $fs_type = 'ext4', # the filesystem type
  Number $size,        # how big
  String $owner     = '0',        # who owns the mount point
  String $group     = '0',        # which group owns the mount point
  Integer $mode      = '0755'     # permissions on mount point
)
```

These are all the parameters for our defined type. Every defined type has to have a `$title` variable defined. An optional `$name` variable can also be defined.

Both `$title` and `$name` are available within the attribute list, so you can specify other attributes using these variables. This is why we can specify our `$mnt` attributes using `$title`. In this case, we'll use the mount point for the filesystem as `$title`, as it should be unique on the node. Any of the previous parameters that are not given a default value, with `=` syntax, must be provided or Puppet will fail catalog compilation with the following error message: `must pass param to Example::Fs[$title] at /path/to/fs.pp:lineno on node nodename.`

Providing sane defaults for parameters means that most of the time you won't have to pass parameters to your defined types, making your code cleaner and easier to read.

Now that we've defined all the parameters required for our filesystem and mounted the combination type, we need to define the type; we can use any of the variables we've asked for as parameters. The definition follows the same syntax as a class definition, as shown:

```
{
  # create the filesystem
  lvm::volume { "$lv":
    ensure => 'present',
    vg     => "$vg",
    pv     => "$pv",
    fstype => "$fs_type",
    size   => "$size",
  }
```

```
# create the mount point ($mnt)
file {"$mnt":
  ensure => 'directory',
  owner  => "$owner",
  group  => "$group",
  mode    => "$mode",
}
# mount the filesystem $lv on the mount point $mnt
mount {"$lv":
  name      => "$mnt",
  ensure    => 'mounted',
  device   => "/dev/$vg/$lv",
  dump     => '1',
  fstype   => "$fs_type",
  options  => "defaults",
  pass     => '2',
  target   => '/etc/fstab',
  require  => [Lvm::Volume["$lv"], File["$mnt"]],
}
}
```

Note that we use the CamelCase notation for requiring `Lvm::Volume` for the mount. CamelCase is the practice of capitalizing each word of a compound word or phrase. This will become useful in the next example where we have nested filesystems that depend on one another. Now, we can redefine our `lvm_web` class using the new `define` to make our intention much clearer, as shown:

```
class lvm_web {
  example::fs {'/var/www/html':
    vg      => 'vg_web',
    lv      => 'lv_var_www',
    pv      => '/dev/sda',
    owner   => '48',
    group   => '48',
    size    => '4G',
    mode    => '0755',
    require  => File['/var/www'],
  }
  file {'/var/www':
    ensure => 'directory',
    mode   => '0755',
  }
}
```

Custom Types

Now, it's clear that we are making sure that the `/var/www` exists for our `/var/www/html` directory to exist and then creating and mounting our filesystem at that point. Now, when we need to make another filesystem on top of `/var/www/html`, we will need to require the first `example::fs` resource. To illustrate this, we will define a subdirectory `/var/www/html/drupal` and require `/var/www/html Example::Fs;` hence, the code becomes easier to follow, as follows:

```
example::fs { '/var/www/html/drupal':
  vg      => 'vg_web',
  lv      => 'lv_drupal',
  pv      => '/dev/sda',
  owner   => '48',
  group   => '48',
  size    => '2G',
  mode    => '0755',
  require  => Example::Fs['/var/www/html']
}
```

The capitalization of `Example::Fs` is important; it needs to be `Example::Fs` for Puppet to recognize this as a reference to the defined type `example::fs`.

Encapsulation makes this sort of chaining much simpler. Also, any enhancements that we make to our defined type are then added to all the instances of it. This keeps our code modular and makes it more flexible. For instance, what if we want to use our `example::fs` type for a directory that may be defined somewhere else in the catalog? We can add a parameter to our definition and set the default value so that the previous uses of the type doesn't cause compilation errors, as shown in the following code:

```
define example::fs
(
  ...
  $managed = true,          # do we create the file resource or not.
  ...
)
```

Now, we can use the `if` condition to create the file and require it (or not), as shown in the following code:

```
if ($managed) {
  file {"$mnt":
    ensure => 'directory',
    owner  => "$owner",
    group  => "$group",
    mode    => "$mode",
  }
```

```
mount {"$lv":  
  name    => "$mnt",  
  ensure   => 'mounted',  
  device   => "/dev/$vg/$lv",  
  dump     => '1',  
  fstype   => "$fs_type",  
  options  => "defaults",  
  pass     => '2',  
  target   => '/etc/fstab',  
  require  => [Lvm::Volume["$lv"],File["$mnt"]],  
}  
}  
} else {  
  mount {"$lv":  
    name    => "$mnt",  
    ensure   => 'mounted',  
    device   => "/dev/$vg/$lv",  
    dump     => '1',  
    fstype   => "$fs_type",  
    options  => "defaults",  
    pass     => '2',  
    target   => '/etc/fstab',  
    require  => Lvm::Volume["$lv"],  
  }  
}  
}
```

None of our existing uses of the `example::fs` type will need modification, but cases where we only want the filesystem to be created and mounted can use this type.

For any portion of code that has repeatable parts, defined types can help abstract your classes to make your meaning more obvious. As another example, we'll develop the idea of an admin user—a user that should be in certain groups, have certain files in their home directory defined, and SSH keys added to their account. The idea here is that your admin users can be defined outside your enterprise authentication system, and only on the nodes to which they have admin rights.

We'll start small using the file and user types to create the users and their home directories. The user has a `managehome` parameter, which creates the home directory but with default permissions and ownership; we'll be modifying those in our type.



If you rely on `managehome`, do understand that `managehome` just passes an argument to the user provider asking the OS-specific tool to create the directory using the default permissions that are provided by that tool. In the case of `useradd` on Linux, the `-m` option is added.

We'll define `~/.bashrc` and `~/.bash_profile` for our user, so we'll need parameters to hold those. An SSH key is useful for admin users, so we'll include a mechanism to include that as well. This isn't an exhaustive solution, just an outline of how you can use `define` to simplify your life. In real world admin scenarios, I've seen the admin define a sudoers file for the admin user and also set up command logging with the audit daemon. Taking all the information we need to define an admin user, we get the following list of parameters:

```
define example::admin
(
    $user = $title,
    $ensure = 'present',
    $uid,
    $home = "/var/home/$title",
    $mode = '0750',
    $shell = "/bin/bash",
    $bashrc = undef,
    $bash_profile = undef,
    $groups = ['wheel', 'bin'],
    $comment = "$title Admin User",
    $expiry = 'absent',
    $forcelocal = true,
    $key,
    $keytype = 'ssh-rsa',
)
```

Now, since `define` will be called multiple times and we need the admin group to exist before we start defining our admin users, we put the group into a separate class and include it, as follows:

```
include example::admin::group
```

The definition of `example::admin::group` is, as follows:

```
class example::admin::group {
    group {'admin':
        gid => 1001,
    }
}
```

With `example::admin::group` included, we move on to define our user, being careful to require the group, as follows:

```
user { "$user":
  ensure      => $ensure,
  allowdupe   => 'true',
  comment     => "$comment",
  expiry      => $expiry,
  forcedlocal => $forcedlocal,
  groups      => $groups,
  home        => $home,
  shell        => $shell,
  uid          => $uid,
  gid          => 1001,
  require     => Group['admin']
}
```

Now, our problem turns to ensuring that the directory containing the home directory exists; the logic here could get very confusing. Since we are defining our admin group by name rather than by `gid`, we need to ensure that the group exists before we create the home directory (so that the permissions can be applied correctly). We are also allowing the home directory location not to exist, so we need to make sure that the directory containing our home directory exists using the following code:

```
# ensure the home directory location exists
$grouprequire = Group['admin']
$dirhome = dirname($home)
```

We are accounting for a scenario where admin users have their home directories under `/var/home`. This example complicates the code somewhat but also shows the usefulness of a defined type.

Since we require the group in all cases, we make a variable hold a copy of that resource definition, as shown in the following code:

```
case $dirhome {
  '/var/home': {
    include example::admin::varhome
    $homerequire = [$grouprequire, File['/var/home']]
  }
}
```

Custom Types

If the home directory is under `/var/home`, we know that the home directory requires the class `example::admin::varhome` and also `File['/var/home']`. Next, if the home directory is under `/home`, then the home directory only needs the group require, as shown in the following code:

```
'/home': {
    # do nothing, included by lsb
    $homerequire = $grouprequire
}
```

As the default option for our case statement, we assume that the home directory needs to require that the directory (`$dirhome`) exists, but the user of this `define` will have to create that resource themselves (`File[$dirhome]`), as follows:

```
default: {
    # rely on definition elsewhere
    $homerequire = [$grouprequire, File[$dirhome]]
}
}
```

Now, we create the home directory using our `$homerequire` variable to define `require` for the resource, as shown:

```
file {"$home":
  ensure  => 'directory',
  owner   => "$uid",
  group   => 'admin',
  mode    => "$mode",
  require  => $homerequire
}
```

Next, we create the `.ssh` directory, as shown:

```
# ensure the .ssh directory exists
file {"$home/.ssh":
  ensure  => 'directory',
  owner   => "$uid",
  group   => 'admin',
  mode    => "0700",
  require  => File["$home"]
}
```

Then, we create an SSH key for the admin user; we require the `.ssh` directory, which requires the home directory, thus making a nice chain of existence. The home directory has to be made first, then the `.ssh` directory, and then the key is added to `authorized_keys`, as shown in the following code:

```
ssh_authorized_key{ "$user-admin":
  user    => "$user",
  ensure  => present,
  type    => "$keytype",
  key     => "$key",
  require  => [User[$user],File["$home/.ssh"]]
}
```

Now we can do something fancy. We know that not every admin likes to work in the same way, so we can have them add custom code to their `.bashrc` and `.bash_profile` files using a concat for the two files. In each case, we'll include the system default file from `/etc/skel` and then permit the instance of the admin user to add to the files using concat, as shown in the following code:

```
# build up the bashrc from a concat
concat { "$home/.bashrc":
  owner => $uid,
  group => $gid,
}
concat::fragment { "bashrc_header_$user":
  target => "$home/.bashrc",
  source => '/etc/skel/.bashrc',
  order  => '01',
}
if $bashrc != undef {
  concat::fragment { "bashrc_user_$user":
    target  => "$home/.bashrc",
    content => $bashrc,
    order   => '10',
  }
}
```

And the same goes for `.bash_profile`, as shown in the following code:

```
#build up the bash_profile from a concat as well
concat { "$home/.bash_profile":
  owner => $uid,
  group => $gid,
}
concat::fragment { "bash_profile_header_$user":
  target => "$home/.bash_profile",
  source => '/etc/skel/.bash_profile',
  order  => '01',
}
if $bash_profile != undef {
  concat::fragment { "bash_profile_user_$user":
    target  => "$home/.bash_profile",
    content => $bash_profile,
    order   => '10',
  }
}
```

We then close our definition with a right brace:

```
}
```

Now, to define an admin user, we call our defined type as shown in the following code and let the type do all the work.

```
example::admin {'theresa':
  uid  => 1002,
  home => '/home/theresa',
  key   => 'BBBB...z',
}
```

We can also add another user easily using the following code:

```
example::admin {'nate':
  uid    => 1001,
  key    => 'AAAA...z',
  bashrc => "alias vi=vim\nexport EDITOR=vim\n"
}
```

Now when we add these resources to a node and run Puppet, we can see the users created:

```

Info: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat::Fragment[bashrc_header_nate]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bashrc/fragments/01_bashrc_header_nate]: Scheduling refresh of Exec[concat/_var/home/nate/.bashrc]
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat::Fragment[bashrc_user_nate]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bashrc/fragments/10_bashrc_user_nate]/ensure: defined content as '{md5}fa76282b2aa138e942cd357b48605eb4'
Info: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat::Fragment[bashrc_user_nate]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bashrc/fragments/10_bashrc_user_nate]: Scheduling refresh of Exec[concat/_var/home/nate/.bashrc]
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bashrc]/Exec[concat/_var/home/nate/.bashrc]/returns: executed successfully
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bashrc]/Exec[concat/_var/home/nate/.bashrc]: Triggered 'refresh' from 4 events
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bashrc]/File[/var/home/nate/.bashrc]/ensure: defined content as '{md5}a347c195093b3bb4026d419f2f12aac7'
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bash_profile]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bash_profile]/ensure: created
Info: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bash_profile]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bash_profile]: Scheduling refresh of Exec[concat/_var/home/nate/.bash_profile]
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bash_profile]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bash_profile/fragments]/ensure: created
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bash_profile]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bash_profile/fragments]: Scheduling refresh of Exec[concat/_var/home/nate/.bash_profile]
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bash_profile]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bash_profile/fragments.concat]/ensure: created
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bash_profile]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bash_profile/fragments.concat.out]/ensure: created
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat::Fragment[bash_profile_header_nate]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bash_profile/fragments/01_bash_profile_header_nate]/ensure: defined content as '{md5}f939eb71a81a9da364410b799e817202'
Info: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat::Fragment[bash_profile_header_nate]/File[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bash_profile/fragments/01_bash_profile_header_nate]: Scheduling refresh of Exec[concat/_var/home/nate/.bash_profile]
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bash_profile]/Exec[concat/_var/home/nate/.bash_profile]/returns: executed successfully
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bash_profile]/Exec[concat/_var/home/nate/.bash_profile]: Triggered 'refresh' from 3 events
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat[/var/home/nate/.bash_profile]/File[/var/home/nate/.bash_profile]/ensure: defined content as '{md5}f939eb71a81a9da364410b799e817202'
Notice: Applied catalog in 0.94 seconds
[root@admins ~]# echo $?
2

```

In this example, we defined a type that created a user and a group, created the user's home directory, added an SSH key to the user, and created their dotfiles. There are many examples where a defined type can streamline your code. Some common examples of defined types include Apache vhosts and Git repositories.

Defined types work well when you can express the thing you are trying to create with the types that are already defined. If the new type can be expressed better with Ruby, then you might have to create your own type by extending Puppet with a custom type.

Types and providers

Puppet separates the implementation of a type into the type definition and any one of the many providers for that type. For instance, the package type in Puppet has multiple providers depending on the platform in use (apt, yum, rpm, gem, and others). Early on in Puppet development there were only a few core types defined. Since then, the core types have expanded to the point where anything that I feel should be a type is already defined by core Puppet. The modules presented in *Chapter 5, Custom Facts and Modules*, created their own types using this mechanism. The LVM module created a type for defining logical volumes, and the concat module created types for defining file fragments. The firewall module created a type for defining firewall rules. Each of these types represents something on the system with the following properties:

- Unique
- Searchable
- Atomic
- Destroyable
- Creatable

When creating a new type, you have to make sure your new type has these properties. The resource defined by the type has to be *unique*, which is why the file type uses the path to a file as the naming variable (`namevar`). A system may have files with the same name (not unique), but it cannot have more than one file with an identical path. As an example, the `ldap` configuration file for `openldap` is `/etc/openldap/ldap.conf`, the `ldap` configuration file for the name services library is `/etc/ldap.conf`. If you used a filename, then they would both be the same resource. Resources must be unique. By *atomic*, I mean it is indivisible; it cannot be made of smaller components. For instance, the `firewall` module creates a type for single iptables rules. Creating a type for the tables (INPUT, OUTPUT, FORWARD) within iptables wouldn't be atomic — each table is made up of multiple smaller parts, the rules. Your type has to be *searchable* so that Puppet can determine the state of the thing you are modifying. A mechanism has to exist to know what the current state is of the thing in question. The last two properties are equally important. Puppet must be able to remove the thing, *destroy* it, and likewise Puppet must be able to *create* the thing anew.

Given these criteria, there are several modules that define new types, with examples including types that manage:

- Git repositories
- Apache virtual hosts
- LDAP entries
- Network routes
- Gem modules
- Perl CPAN modules
- Databases
- Drupal multisites

Creating a new type

As an example, we will create a gem type for managing Ruby gems installed for a user. Ruby gems are packages for Ruby that are installed on the system and can be queried like packages.



Installing gems with Puppet can already be done using the `gem`, `pe_gem`, or `pe_puppetserver_gem` providers for the package type.

Creating a custom type requires some knowledge of Ruby. In this example, we assume the reader is fairly literate in Ruby. We start by defining our type in the `lib/puppet/type` directory of our module. We'll do this in our example module, `modules/example/lib/puppet/type/gem.rb`.

The file will contain the `newtype` method and a single property for our type, `version`, as shown in the following code:

```
Puppet::Type.newtype(:gem) do
  ensurable
  newparam(:name, :namevar => true) do
    desc 'The name of the gem'
  end
  newproperty(:version) do
    desc 'version of the gem'
    validate do |value|
      fail("Invalid gem version #{value}") unless value =~
/^([0-9]+|[0-9A-Za-z\.-]+)$/
    end
  end
end
```

The `ensurable` keyword creates the `ensure` property for our new type, allowing the type to be either present or absent. The only thing we require of the version is that it starts with a number and only contain numbers, letters, periods, or dashes.



A more thorough regular expression here could save you time later, such as checking that the version ends with a number or letter.



Now we need to start making our provider. The name of the provider is the name of the command used to manipulate the type. For packages, the providers have names such as `yum`, `apt`, and `dpkg`. In our case we'll be using the `gem` command to manage gems, which makes our path seem a little redundant. Our provider will live at `modules/example/lib/puppet/provider/gem/gem.rb`.

We'll start our provider with a description of the provider and the commands it will use are, as shown here:

```
Puppet::Type.type(:gem).provide :gem do
  desc "Manages gems using gem"
```

Then we'll define a method to list all the gems installed on the system as shown here, which defines the `self.instances` method:

```
def self.instances
  gems = []
  command = 'gem list -l'
  begin
    stdin, stdout, stderr = Open3.popen3(command)
    for line in stdout.readlines
      (name,version) = line.split(' ')
      gem = {}
      gem[:provider] = self.name
      gem[:name] = name
      gem[:ensure] = :present
      gem[:version] = version.tr('()','')
      gems << new(gem)
    end
  rescue
    raise Puppet::Error, "Failed to list gems using '#{command}'"
  end
  gems
end
```

This method runs `gem list -l` and then parses the output, looking for lines such as `gemname (version)`. The output from the `gem` command is written to the variable `stdout`. We then use `readlines` on `stdout` to create an array that we iterate over with a `for` loop. Within the `for` loop we split the lines of output based on a space character into the gem name and version. The version will be wrapped in parentheses at this point; we use the `tr` (translate) method to remove the parentheses. We create a local hash of these values and then append the hash to the `gems` hash. The `gems` hash is returned and then Puppet knows all about the gems installed on the system.

Puppet needs two more methods at this point, a method to determine if a gem exists (is installed), and, if it does exist, one to tell us which version is installed. We already populated the `ensure` parameter, so as to use that to define our `exists` method as follows:

```
def exists?
  @property_hash[:ensure] == :present
end
```

To determine the version of an installed gem, we can use the `property_hash` variable, as follows:

```
def version
  @property_hash[:version] || :absent
end
```

To test this, add the module to a node and `pluginsync` the module over to the node, as shown:

```
[root@client ~]# puppet plugin download
Notice: /File[/opt/puppetlabs/puppet/cache/lib/puppet/provider/gem]/
ensure: created
Notice: /File[/opt/puppetlabs/puppet/cache/lib/puppet/provider/gem/gem.
rb]/ensure: defined content as '{md5}4379c3d0bd6c696fc9f9593a984926d3'
Notice: /File[/opt/puppetlabs/puppet/cache/lib/puppet/
provider/gem/gem.rb.orig]/ensure: defined content as '{md5}
c6024c240262f4097c0361ca53c7bab0'
Notice: /File[/opt/puppetlabs/puppet/cache/lib/puppet/type/gem.rb]/
ensure: defined content as '{md5}48749efcd33ce06b401d5c008d10166c'
Downloaded these plugins: /opt/puppetlabs/puppet/cache/lib/puppet/
provider/gem, /opt/puppetlabs/puppet/cache/lib/puppet/provider/gem/gem.
rb, /opt/puppetlabs/puppet/cache/lib/puppet/provider/gem/gem.rb.orig, /
opt/puppetlabs/puppet/cache/lib/puppet/type/gem.rb
```

Custom Types

This will install our type/gem.rb and provider/gem/gem.rb files into /opt/puppetlabs/puppet/cache/lib/puppet on the node. After that, we are free to run puppet resource on our new type to list the available gems, as shown:

```
[root@client ~]# puppet resource gem
gem { 'bigdecimal':
  ensure  => 'present',
  version => '1.2.0',
}
gem { 'bropages':
  ensure  => 'present',
  version => '0.1.0',
}
gem{ 'commander':
  ensure  => 'present',
  version => '4.1.5',
}
gem { 'highline':
  ensure  => 'present',
  version => '1.6.20',
}
...
...
```

Now, if we want to manage gems, we'll need to create and destroy them, and we'll need to provide methods for those operations. If we try at this point, Puppet will fail, as we can see from the following output:

```
[root@client ~]# puppet resource gem bropages
gem { 'bropages':
  ensure  => 'present',
  version => '0.1.0',
}
[root@client ~]# puppet resource gem bropages ensure=absent
gem { 'bropages':
  ensure => 'absent',
}
[root@client ~]# puppet resource gem bropages ensure=absent
```

```
gem { 'bropages':
  ensure => 'absent',
}
```

When we run `puppet resource`, there is no `destroy` method, so Puppet returns that the gem was removed but doesn't actually do anything. To get Puppet to actually remove the gem, we'll need a method to destroy (remove) gems; `gem uninstall` should do the trick, as shown in the following code:

```
def destroy
  g = @resource[:version] ? [@resource[:name], '--version', @
  resource[:version]] : @resource[:name]
  command = "gem uninstall #{g} -q -x"
  begin
    system command
  rescue
    raise Puppet::Error, "Failed to remove #{@resource[:name]} "
  '#{command}'"
  end
  @property_hash.clear
end
```

Using the ternary operator, we either run `gem uninstall name -q -x` if no version is defined, or `gem uninstall name --version version -q -x` if a version is defined. We finish by calling `@property_hash.clear` to remove the gem from the `property_hash` since the gem is now removed.

Now we need to let Puppet know about the state of the `bropages` gem using the `instances` method we defined earlier; we'll need to write a new method to prefetch all the available gems. This is done with `self.prefetch`, as shown here:

```
def self.prefetch(resources)
  gems = instances
  resources.keys.each do |name|
    if provider = gems.find{ |gem| gem.name == name }
      resources[name].provider = provider
    end
  end
end
```

We can see this in action using `puppet resource` as shown here:

```
[root@client ~]# puppet resource gem bropages ensure=absent
Removing bro
Successfully uninstalled bropages-0.1.0
```

Custom Types

```
Notice: /Gem[bropages]/ensure: removed
gem { 'bropages':
  ensure => 'absent',
}
```

Almost there! Now we want to add bropages back, we'll need a `create` method, as shown here:

```
def create
  g = @resource[:version] ? [@resource[:name], '--version', @_resource[:version]] : @resource[:name]
  command = "gem install #{g} -q"
begin
  system command
  @property_hash[:ensure] = :present
rescue
  raise Puppet::Error, "Failed to install #{@resource[:name]} "
'#{command}'"
end
end
```

Now, when we run `puppet resource` to create the gem, we see the installation, as shown here:

```
[root@client ~]# puppet resource gem bropages ensure=present
Successfully installed bropages-0.1.0
Parsing documentation for bropages-0.1.0
Installing ri documentation for bropages-0.1.0
1 gem installed
Notice: /Gem[bropages]/ensure: created
gem { 'bropages':
  ensure => 'present',
}
```

Nearly done! Now, we need to handle versions. If we want to install a specific version of the gem, we'll need to define methods to deal with versions.

```
def version=(value)
  command = "gem install #{@resource[:name]} --version #{@_resource[:version]}"
begin
  system command
```

```
    @property_hash[:version] = value
rescue
  raise Puppet::Error, "Failed to install gem #{resource[:name]} "
using "#{command}"
end
end
```

Now, we can tell Puppet to install a specific version of the gem and have the correct results as shown in the following output:

```
[root@client ~]# puppet resource gem bropages version='0.0.9'
Fetching: highline-1.7.8.gem (100%)
Successfully installed highline-1.7.8
Fetching: bropages-0.0.9.gem (100%)
Successfully installed bropages-0.0.9
Parsing documentation for highline-1.7.8
Installing ri documentation for highline-1.7.8
Parsing documentation for bropages-0.0.9
Installing ri documentation for bropages-0.0.9
2 gems installed
Notice: /Gem[bropages]/version: version changed '0.1.0' to '0.0.9'
gem { 'bropages':
  ensure => 'present',
  version => '0.0.9',
}
```

This is where our choice of gem as an example breaks down as gem provides for multiple versions of a gem to be installed. Our gem provider, however, works well enough for use at this point. We can specify the gem type in our manifests and have gems installed or removed from the node. This type and provider are only an example; the gem provider for the package type provides the same features in a standard way. When considering creating a new type and provider, search Puppet Forge for existing modules first.

Summary

It is possible to increase the readability and resiliency of your code using parameterized classes and defined types. Encapsulating sections of your code within a defined type makes your code more modular and easier to support. When the defined types are not enough, you can extend Puppet with custom types and providers written in Ruby. The details of writing providers are best learned by reading the already written providers and referring to the documentation on the Puppet Labs website. The public modules covered in an earlier chapter make use of defined types, custom types and providers, and can also serve as a starting point to write your own types. The `augeasproviders` module is another module to read when looking to write your own types and providers.

In the next chapter, we will set up reporting and look at Puppet Dashboard and the Foreman.

7

Reporting and Orchestration

Reports return all the log messages from Puppet nodes to the master. In addition to log messages, reports send other useful metrics such as timing (time spent performing different operations) and statistical information (counts of resources and the number of failed resources). With reports, you can know when your Puppet runs fail and, most importantly, why. In this chapter, we will cover the following reporting mechanisms:

- Syslog
- Store (YAML)
- IRC
- Foreman
- Puppet Dashboard

In addition to reporting, we will configure the **marionette collective (mcollective)** system to allow for orchestration tasks. In the course of configuring reporting, we will show different methods of signing and transferring SSL keys for systems that are subordinate to our master, `puppet.example.com`.

Turning on reporting

To turn on reporting, set `report = true` in the `[agent]` section of `puppet.conf` on all your nodes.

Once you have done that, you need to configure the master to deal with reports. There are several report types included with Puppet; they are listed at: <http://docs.puppetlabs.com/references/latest/report.html>. Puppet Labs documentation on reporting can be found at: <http://docs.puppetlabs.com/guides/reporting.html>.

There are three simple reporting options included with Puppet: `http`, `log`, and `store`. The `http` option will send the report as a YAML file via a POST operation to the HTTP or HTTPS URL pointed to by the `reporturl` setting in `puppet.conf`. The `log` option uses syslog to send reports from the nodes via syslog on the master; this method will only work with the WEBrick and Passenger implementations of Puppet. `puppetserver` sends syslog messages via the Logback mechanism, which is covered in a following section. The last option is `store`, which simply stores the report as a file in `reportdir` of the master.

To use a report, add it by name to the `reports` section on the master. This is a comma-separated list of reports. You can have many different report handlers. Report handlers are stored at `site_ruby/[version]/puppet/reports/` and `/var/lib/puppet/lib/puppet/reports`. The latter directory is where modules can send report definitions to be installed on clients (using the `pluginsync` mechanism; remember that things get purged from the `pluginsync` directories so, unless you are placing files there with Puppet, they will be removed).

Store

To enable the store mechanism, use `reports = store`. We'll add this to our log destination in this example, as shown in the following snippet:

```
[main]
reports = store
```

The default location for `reports` is `reportdir`. To see your current `reportdir` directory, use the `--configprint` option on the master, as shown in the following snippet:

```
[root@stand ~]# puppetconfig print reportdir
/opt/puppetlabs/server/data/puppetserver/reports
```

The `store` option is on by default; however, once you specify the `reports` setting as anything in the `main` section of `puppet.conf`, you disable the implicit `store` option. Remember that report files will start accumulating on the master. It's a good idea to enable purging of those reports. In our multiple-master scenario, it's a good idea to set `report_server` in the `agent` section of the nodes if you are using `store`, as shown in the following commands. The default setting for `report_server` is the same as the `server` parameter:

```
[root@client ~]# puppetconfig print report_server
report_server = puppet
server = puppet
```

After enabling reports on the client and `reports = store` on the server, you will begin seeing reports in the `reportdir` directory, as shown here:

```
[root@stand ~]# puppetconfig print reportdir
/opt/puppetlabs/server/data/puppetserver/reports
[root@stand ~]# ls /opt/puppetlabs/server/data/puppetserver/reports/
client.example.com/
201509130433.yaml 201509160551.yaml 201509252128.yaml 201510031025.
yaml 201510040502.yaml
...
```

In the next section, we will look at the logging configuration of `puppetserver`.

Logback

Due to `puppetserver` running as a JRuby instance within a JVM, Java's logback mechanism is used for logging. Logback is configured in the `logback.xml` file in the `/etc/puppetlabs/puppetserver` directory. The default log level is `INFO` and is specified within the `<logger>` XML entity; it may be changed to `DEBUG` or `TRACE` for more information. `puppetserver` directs its logs to the `/var/log/puppetlabs/puppetserver/puppetserver.log` file, as specified in the `<appender>` XML entity. More information on logback is available at <http://logback.qos.ch/>.

In the next section we will look at one of the community-supported reporting plugins, a plugin for IRC.

Internet relay chat

If you have an internal **Internet Relay Chat (IRC)** server, using the IRC report plugin can be useful. This report sends failed catalog compilations to an IRC chat room. You can have this plugin installed on all your catalog workers; each catalog worker will log in to the IRC server and send failed reports. That works very well, but in this example we'll configure a new worker called `reports.example.com`. It will be configured as though it were a standalone master; the `reports` machine will need the same package as a regular master (`puppetserver`). We'll enable the IRC logging mechanism on this server. That way we only have to install the dependencies for the IRC reporter on one master.

The reports server will need certificates signed by `puppet.example.com`. There are two ways you can have the keys created; the simplest way is to make your reports server a client node of `puppet.example.com` and have Puppet generate the keys. We will show how to use the `puppet certificate generate` command to manually create and download keys for our reports server.

First, generate certificates for this new server on `puppet.example.com` using `puppet certificate generate`.

 The `puppet certificate generate` command may be issued from either `puppet.example.com` or `reports.example.com`. When running from `puppet.example.com`, the command looks as follows:

```
# puppet certificate generate --ca-location local  
reports.example.com
```

When running from `reports.example.com`, the command looks as follows:

```
# puppet certificate generate --ca-location remote  
--server puppet.example.com reports.example.com
```

You will then need to sign the certificate on `puppet.example.com` using the following command:

```
[root@stand ~]# puppet cert sign reports.example.com  
Log.newmessage notice 2015-11-15 20:42:03 -0500 Signed certificate  
request for reports.example.com  
Notice: Signed certificate request for reports.example.com  
Log.newmessage notice 2015-11-15 20:42:03 -0500 Removing file Puppet::SS  
L::CertificateRequestreports.example.com at '/etc/puppetlabs/puppet/ssl/  
ca/requests/reports.example.com.pem'  
Notice: Removing file Puppet::SSL::CertificateRequestreports.example.com  
at '/etc/puppetlabs/puppet/ssl/ca/requests/reports.example.com.pem'
```

If you used `puppet certificate generate`, then you will need to download the public and private keys from `puppet.example.com` to `reports.example.com`. The private key will be in `/etc/puppetlabs/puppet/ssl/private_keys/reports.
example.com.pem`, and the public key will be in `/etc/puppetlabs/puppet/ssl/
ca/signed/reports.example.com.pem`.

We can use `puppet certificate` to do this as well. On the reports machine, run the following command:

```
[root@reports ~]# puppet certificate find reports.example.com --ca-
location remote --server puppet.example.com
-----BEGIN CERTIFICATE-----
...
eCXSPKRz/0mz0q/xDD+Zy8yU
-----END CERTIFICATE-----
```

The report machine will need the certificate authority files as well (`/etc/puppetlabs/puppet/ssl/ca/ca_crt.pem` and `/etc/puppetlabs/puppet/ssl/ca/ca_crl.pem`); the **Certificate Revocation List (CRL)** should be kept in sync using an automated mechanism. The CRL is used when certificates are invalidated with the `puppet certificate destroy`, `puppet cert clean`, or `puppet cert revoke` commands.

To download the CA from `puppet.example.com`, use the following command:

```
[root@reports ~]# puppet certificate find ca --ca-location remote
--server puppet.example.com
-----BEGIN CERTIFICATE-----
...
```

The CRL will have to be downloaded manually.

By default, the `puppetserver` service will attempt to run the built-in CA and sign certificates; we don't want our report server to do this, so we need to disable the CA service in `/etc/puppetlabs/puppetserver/bootstrap.cfg` by following the instructions given in the file as shown here:

```
# To enable the CA service, leave the following line uncommented
#puppetlabs.services.ca.certificate-authority-service/certificate-
authority-service
# To disable the CA service, comment out the above line and uncomment
the line below
puppetlabs.services.ca.certificate-authority-disabled-service/
certificate-authority-disabled-service
```

Next we need to add some certificate settings to the `webserver.conf` file within the `/etc/puppetlabs/puppetserver/conf.d` directory, as shown here:

```
ssl-cert = /etc/puppetlabs/puppet/ssl/certs/reports.example.com.pem
ssl-key = /etc/puppetlabs/puppet/ssl/private_keys/reports.example.com.
pem
ssl-ca-cert = /etc/puppetlabs/puppet/ssl/certs/ca.pem
```

Now you can run Puppet on your nodes that are configured to send reports to `report_server=reports.example.com`, and the reports will show up in `$reportdir`. With report forwarding in place, we'll turn to installing the IRC plugin. First use `puppet module` to install the module:

```
[root@reports ~]# puppet module install jamtur01/irc
Log.newmessage notice 2015-11-15 22:19:53 -0500 Preparing to install into
/etc/puppetlabs/code/environments/production/modules ...
Notice: Preparing to install into /etc/puppetlabs/code/environments/
production/modules ...
Log.newmessage notice 2015-11-15 22:19:53 -0500 Downloading from https://
forgeapi.puppetlabs.com ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Log.newmessage notice 2015-11-15 22:19:56 -0500 Installing -- do not
interrupt ...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/code/environments/production/modules
└─ jamtur01-irc (v0.0.7)
    └─ puppetlabs-stdlib (v4.9.0)

[root@reports ~]# cp /etc/puppetlabs/code/environments/production/
modules/irc/lib/puppet/reports/irc.rb /opt/puppetlabs/puppet/lib/ruby/
vendor_ruby/puppet/reports/
```



Search for `puppet/reports` to find the reports directory.



Now copy the `irc.yaml` configuration file into `/etc/puppetlabs`, and edit it as appropriate. Our IRC server is `irc.example.com`. We'll use the username `puppetbot` and password `PacktPubBot`, as shown in the following snippet:

```
---
:irc_server: 'irc://puppetbot:PacktPubBot@irc.example.com:6667#puppet'
:irc_ssl: false
:irc_register_first: false
:irc_join: true
:report_url: 'http://foreman.example.com/hosts/%h/reports/last'
```

We are almost ready; the IRC report plugin uses the `carrier-pigeon` Ruby gem to do the IRC work, so we'll need to install that now. Since reports run within the `puppetserver` process, we need to install the gem within `puppetserver`, as shown here:

```
[root@reports ~]# puppetserver gem install carrier-pigeon
Fetching: addressable-2.3.8.gem (100%)
Successfully installed addressable-2.3.8
Fetching: carrier-pigeon-0.7.0.gem (100%)
Successfully installed carrier-pigeon-0.7.0
2 gems installed
```

Now we can restart `puppetserver` on our reports worker and create a catalog compilation problem on the client. To ensure the catalog fails to compile, I've edited `site.pp` and added the following line to the default node definition:

```
fail('fail for no good reason')
```

This causes the catalog to fail compilation on our client node as shown in the following screenshot:

```
[root@client ~]# puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Error: Could not retrieve catalog from remote server: Error 400 on SERVER: Evaluation Error: Error while evaluating a Function Call, fail for no good reason at /etc/puppetlabs/code/environments/production/manifests/site.pp:20:3 on node client.example.com
Warning: Not using cache on failed catalog
Error: Could not retrieve catalog; skipping run
```

Whenever a catalog fails to compile, the IRC report processor will log in to our `#puppet` channel as the `puppetbot` user and let us know, as shown in the following IRSSI (IRC client) screenshot:

```
15:40 < puppetbot> Puppet production run for client.example.com failed at Mon Nov 16 15:40:31
2015. Report URL:
http://foreman.example.com/hosts/client.example.com/reports/last
15:40 -!- puppetbot [~puppetbot@192.168.1.5] has quit []
```

Now for our next task, the given URL requires that Foreman is configured; we'll set up that now.

Foreman

Foreman is more than just a Puppet reporting tool; it bills itself as a complete life cycle management platform. Foreman can act as the **external node classifier (ENC)** for your entire installation and configure DHCP, DNS, and PXE booting. It's a one-stop shop. We'll configure Foreman to be our report backend in this example.

Installing Foreman

To install Foreman, we'll need **Extra Packages for Enterprise Linux (EPEL)** (<https://fedoraproject.org/wiki/EPEL>) and **Software Collections (SCL)** (<https://fedorahosted.org/SoftwareCollections/>), which are the yum repositories for Ruby 1.9.3 and its dependencies. We have previously used the EPEL repository; the SCL repository is used for updated versions of packages that already exist on the system, in this case, Ruby 1.9.3 (Ruby 2.0 is the default on Enterprise Linux 7). The SCL repositories have updated versions of other packages as well. To install EPEL and SCL, use the following package locations:

- <https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm>
- http://yum.theforeman.org/releases/1.9/el7/x86_64/rhscl-ruby193-epel-7-x86_64-1-2.noarch.rpm

With these two repositories enabled, we can install Foreman using the Foreman yum repository as shown here:

```
# yum -y install http://yum.theforeman.org/releases/latest/el7/x86_64/
foreman-release.rpm
# yum -y install foreman-installer
```

The `foreman-installer` command uses `puppet apply` to configure Foreman on the server. Since we will only be using Foreman for reporting in this example, we can just use the installer, as shown here:

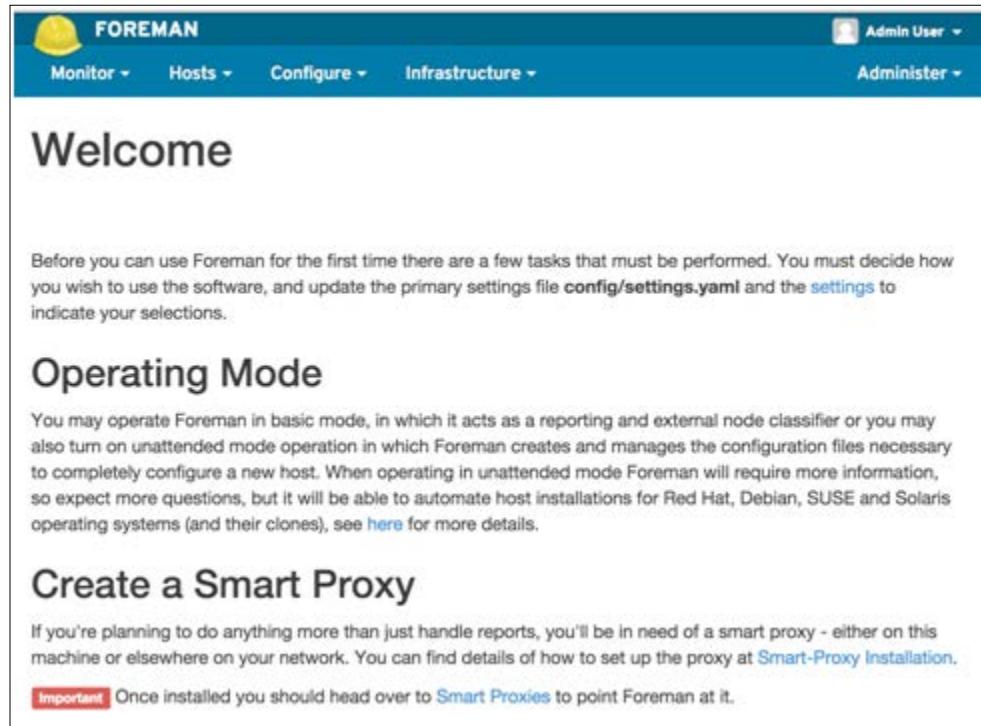
```
[root@foreman ca]# foreman-installer --no-enable-foreman-proxy --no-
enable-puppet --puppet-ca-server puppet.example.com
WARN: Unresolved specs during Gem::Specification.reset:
multi_json(>= 1.8.4)
WARN: Clearing out unresolved specs.
Please report a bug if this causes problems.

Installing           Done
[100%] [.....]
Success!
```

```
* Foreman is running at https://foreman.example.com
Initial credentials are admin / ppNmZefciG6HxU4q
The full log is at /var/log/foreman-installer/foreman-installer.log
```

The installer will pull down all the Ruby gems required for Foreman and install and configure PostgreSQL by default. The database will be populated and started using `puppet apply`. The Foreman web application will be configured using `mod_passenger` and Apache.

At this point, you can connect to Foreman and log in using the credentials given in the output. The password is automatically created and is unique to each installation. The main screen of Foreman is shown in the following screenshot:



Attaching Foreman to Puppet

With Foreman installed and configured, create certificates for `foreman.example.com` on `puppet.example.com`, and copy the keys over to Foreman; they will go in `/var/lib/puppet/ssl` using the same procedure as we did for `reports.example.com` at the beginning of the chapter.

We need our report server to send reports to Foreman, so we need the `foreman-report` file. You can download this from https://raw.githubusercontent.com/theforeman/puppet-foreman/master/files/foreman-report_v2.rb or use the one that `foreman-installer` installed for you. This file will be located in: `/usr/share/foreman-installer/modules/foreman/files/foreman-report_v2.rb`.

Copy this file to `reports.example.com` in `/opt/puppetlabs/puppet/lib/ruby/vendor_ruby/puppet/reports/foreman.rb`. Create the Foreman configuration file in `/etc/puppet/foreman.yaml`, and create the `/etc/puppet` directory if it does not exist. The contents of `foreman.yaml` should be the following:

```
---
# Update for your Foreman and Puppet master hostname(s)
:url: "https://foreman.example.com"
:ssl_ca: "/etc/puppetlabs/puppet/ssl/certs/ca.pem"
:ssl_cert: "/etc/puppetlabs/puppet/ssl/certs/reports.example.com.pem"
:ssl_key: "/etc/puppetlabs/puppet/ssl/private_keys/reports.example.com.pem"

# Advanced settings
:puppetdir: "/opt/puppetlabs/puppet/cache"
:puppetuser: "puppet"
:facts: true
:timeout: 10
:threads: null
```

Next, add Foreman to the `reports` line in `puppet.conf` and restart `puppetserver`. So far we have our Puppet nodes sending reports to our reporting server, which is in turn sending reports to Foreman. Foreman will reject the reports at this point until we allow `reports.example.com`. Log in to <https://foreman.example.com> using the admin account and password.

Then navigate to the **Settings** section under **Administer** as shown in the following screenshot:



Click on the **Auth** tab, and update the **trusted_puppetmaster_hosts** setting:



Note that this must be an array, so keep the [] brackets around `reports.example.com`, as shown in the following screenshot:



With all this in place, when a node compiles a catalog, it will send the report to `reports.example.com`, which will send the report on to `foreman.example.com`. After a few reports arrive, our Foreman homepage will list hosts and reports.

Using Foreman

Let's first look at the **Hosts** window shown in the following screenshot:

A screenshot of the Foreman 'Hosts' window. The top navigation bar shows 'Monitor', 'Hosts', 'Configure', 'Infrastructure', and 'Administer'. The 'Hosts' tab is selected. The main area is titled 'Hosts' and contains a table with four rows. The columns are: Name, Operating system, Environment, Model, Host group, and Last report. The table data is as follows:

	Name	Operating system	Environment	Model	Host group	Last report	
<input type="checkbox"/>	foreman.example.com					less than a minute ago	<button>Edit</button>
<input type="checkbox"/>	node1.example.com					6 minutes ago	<button>Edit</button>
<input type="checkbox"/>	reports.example.com					2 minutes ago	<button>Edit</button>
<input type="checkbox"/>	worker1.example.com					3 minutes ago	<button>Edit</button>

Below the table, a message says 'Displaying all 4 entries - 0 selected'. There is also a green 'New Host' button in the top right corner.

Reporting and Orchestration

The icons next to the hostnames indicate the status of the last Puppet run. You can also navigate to the **Monitor | Reports** section to see the latest reports, as shown in the following screenshot:

The screenshot shows the Foreman interface with the title "Reports". At the top, there is a search bar with the query "eventful = true" and a "Search" button. Below the search bar is a table with the following columns: Host, Last report, Applied, Restarted, Failed, Restart Failures, Skipped, Pending, and an empty column for actions. The table contains the following data:

Host	Last report	Applied	Restarted	Failed	Restart Failures	Skipped	Pending	
foreman.example.com	3 minutes ago	2	0	0	0	0	0	Delete
reports.example.com	5 minutes ago	3	0	0	0	0	0	Delete
reports.example.com	5 minutes ago	0	0	1	0	0	0	Delete
worker1.example.com	6 minutes ago	0	0	1	0	0	0	Delete
node1.example.com	8 minutes ago	2	0	0	0	0	0	Delete

At the bottom of the table, a message says "Displaying all 5 entries".

Clicking on `client.example.com` shows the failed catalog run and the contents of the error message, as shown in the following screenshot:

The screenshot shows a report for a Puppet run. The report is titled "Puppet" and contains the following message: "Could not retrieve catalog from remote server: Error 400 on SERVER: Evaluation Error: Error while evaluating a Function Call, fail for no good reason at /etc/puppetlabs/code/environments/production/manifests/site.pp:20:3 on node client.example.com".

Another great feature of Foreman is that, when a file is changed by Puppet, Foreman will show the `diff` file for the change in a pop-up window. When we configured our IRC bot to inform us of failed Puppet runs in the last section, the bot presented URLs for reports; those URLs were Foreman-specific and will now work as intended. The Foreman maintainers recommend purging your Puppet reports to avoid filling the database and slowing down Foreman. They have provided a rakefile that can be run with `foreman-rake` to delete old reports, as shown here:

```
[root@foreman ~]# foreman-rake reports:expire days=7
```

To complete this example, we will have our master facts sent to Foreman. This is something that can be run from cron. Copy the `node.rb` ENC script from https://raw.githubusercontent.com/theforeman/puppet-foreman/2.2.3/files/external_node_v2.rb to the `stand.example.com` Puppet master.

Copy the `foreman.yaml` configuration file from `reports.example.com` to `stand.example.com`. Again, go back into the Foreman GUI and add `stand.example.com` to `trusted_puppetmaster_hosts`. Then, from `stand` run the `node.rb` script with `--push-facts` to push all the facts to Foreman, as shown here:

```
[root@stand ~]# /etc/puppet/node.rb --push-facts
```

Now, when you view hosts in Foreman, they will have their facts displayed. Foreman also includes rakefiles to produce e-mail reports on a regular basis. Information on configuring these is available at: http://projects.theforeman.org/projects/foreman/wiki/Mail_Notifications.

With this configuration, Foreman is only showing us the reports. Foreman can be used as a full ENC implementation and take over the entire life cycle of provisioning hosts. I recommend looking at the documentation and exploring the GUI to see if you might benefit from using more of Foreman's features.

Puppet GUIs

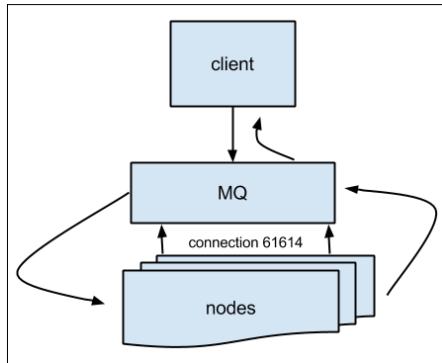
Representing Puppet report information in a web GUI is a useful idea. There are several GUIs available; Puppet Labs has Puppet Enterprise and its console interface. Other open source alternatives are **Puppetboard** (<https://github.com/voxpupuli/puppetboard>), **PanoPuppet** (<https://github.com/propyless/panopuppet>), and **Puppet Explorer** (<https://github.com/spotify/puppetexplorer>). All these tools rely on PuppetDB for their data. These tools are developing quickly and changing, so I suggest trying each one and finding the one that offers the features best suited to your needs.

mcollective

mcollective is an orchestration tool created by Puppet Labs that is not specific to Puppet. Plugins exist to work with other configuration management systems. mcollective uses a **Message Queue (MQ)** tool with active connections from all active nodes to enable parallel job execution on a large numbers of nodes.

To understand how mcollective works, we'll consider the following high-level diagram and work through various components. The configuration of mcollective is somewhat involved and prone to errors. Still, once mcollective is working properly, the power it provides can become addictive. It will be worth the effort, I promise.

In the following diagram, we see that the client executing the `mcollective` command communicates with the MQ server. The MQ server then sends the query to each of the nodes connected to the queue.



The default MQ installation for marionette uses `activemq`. The `activemq` package provided by the Puppet Labs repository is known to work.

 mcollective uses a generic message queue and can be configured to use your existing message queue infrastructure.

If using `activemq`, a single server can handle 800 nodes. After that, you'll need to spread out to multiple MQ servers. We'll cover the standard mcollective installation using Puppet's certificate authority to provide SSL security to mcollective. The theory here is that we trust Puppet to configure the machines already; we can trust it a little more to run arbitrary commands. We'll also require that users of mcollective have proper SSL authentication.

 You can install mcollective using the `mcollective` module from Forge (<https://forge.puppetlabs.com/puppetlabs/mcollective>). In this section, we will install mcollective manually to explain the various components.

Installing ActiveMQ

ActiveMQ is the recommended messaging server for mcollective. If you already have a messaging server in your infrastructure, you can use your existing server and just create a message queue for mcollective. To install ActiveMQ, we'll use a different Puppet Labs repository than we used to install Puppet; this repository is located at <http://yum.puppetlabs.com/puppetlabs-release-el-7.noarch.rpm>:

1. We install ActiveMQ from the Puppet Labs repository to `puppet.example.com` using the following command:

```
# yum install activemq
...
Installed:
activemq.noarch0:5.9.1-2.el7
```

2. Next, download the sample ActiveMQ config file using the following commands:

```
[root@stand ~]# cd /etc/activemq
[root@standactivemq]# mv activemq.xml activemq.xml.orig
[root@standactivemq]# curl -O https://raw.githubusercontent.com/
puppetlabs/marionette-collective/master/ext/activemq/examples/
single-broker/activemq.xml
```

3. This will create `activemq.xml`. This file needs to be owned by the user `activemq` and, since we will be adding passwords to the file shortly, we'll set its access permissions to user-only:

```
[root@standactivemq]# chown activemq activemq.xml
[root@standactivemq]# chmod 0600 activemq.xml
```

4. Now create an mcollective password and admin password for your message queue using the following code. The defaults in this file are `marionette` and `secret` respectively:

```
<simpleAuthenticationPlugin>
<users>
<authenticationUser username="mcollective"
password="PacktPubSecret" groups="mcollective, everyone"/>
<authenticationUser username="admin"
password="PacktPubSuperSecret" groups="mcollective, admins, everyone"/>
</users>
</simpleAuthenticationPlugin>
```

5. Next, change the `transportConnectors` section to use SSL, as shown in the following snippet:

```
<transportConnectors>
<transportConnector name="openwire" uri="tcp://0.0.0.0:61616"/>
<transportConnector name="stomp+ssl" uri="stomp+ssl://0.0.0.0:6161
4?needClientAuth=true"/>
</transportConnectors>
```

6. Immediately following the `transportConnectors`, we'll define an `sslContext`, which will contain the SSL keys from our Puppet master in a format compatible with ActiveMQ (keystores):

```
<sslContext>
  <sslContext
keyStore="keystore.jks" keyStorePassword="PacktPubKeystore"
trustStore="truststore.jks" trustStorePassword="PacktPubTrust"
/>
</sslContext>
```

This section should be within the `<broker>` definition. For simplicity, just stick it right after the `<transportConnectors>` section.

7. Now we need to create `keystore.jks` and `truststore.jks`. Start by copying the certificates from Puppet into a temporary directory, as shown here:

```
[root@stand ~]# cd /etc/activemq
[root@standactivemq]# mkdir tmp
[root@standactivemq]# cd tmp
[root@standtmp]# cp /etc/puppetlabs/puppet/ssl/certs/ca.pem .
[root@standtmp]# cp /etc/puppetlabs/puppet/ssl/certs/puppet.
example.com.pem .

[root@standtmp]# cp /etc/puppetlabs/puppet/ssl/private_keys/
puppet.example.com.pem puppet.example.com.private.pem

[root@standtmp]# keytool -import -alias "Example CA" -file ca.pem
-keystore truststore.jks

Enter keystore password: PacktPubTrust
Re-enter new password: PacktPubTrust
Owner: CN=Puppet CA: puppet.example.com
Issuer: CN=Puppet CA: puppet.example.com
...
Trust this certificate? [no]: yes
Certificate was added to keystore
```

- Now that the truststore.jks keystore is complete, we need to create the keystore.jks keystore. We start by combining the public and private portions of the puppetserver certificate. The combined file is then fed to OpenSSL's pkcs12 command to create a pkcs12 file suitable for import using keytool:

```
[root@standtmp]# cat puppet.example.com.pem puppet.example.com.private.pem > puppet.pem
[root@standtmp]# openssl pkcs12 -export -in puppet.pem -out activemq.p12 -name puppet.example.com
Enter Export Password: PacktPubKeystore
Verifying - Enter Export Password: PacktPubKeystore
[root@standtmp]# keytool -importkeystore -destkeystore keystore.jks -srckeystore activemq.p12 -srcstoretype PKCS12 -alias puppet.example.com
Enter destination keystore password: PacktPubKeystore
Re-enter new password: PacktPubKeystore
Enter source keystore password: PacktPubKeystore
```

- Now these files are created, so move them into /etc/activemq, and make sure they have the appropriate permissions:

```
[root@standtmp]# chown activemq truststore.jks keystore.jks
[root@standtmp]# chmod 0600 truststore.jks keystore.jks
[root@standtmp]# mv truststore.jks keystore.jks /etc/activemq/
```



The ActiveMQ rpm is missing a required symlink; ActiveMQ will not start until /usr/share/activemq/activemq-data is symlinked to /var/cache/activemq/data.



- We can now start activemq using the following command; make sure that your firewall allows connections inbound on port 61614, which is the port specified in the transportConnector line in activemq.xml:

```
[root@stand ~]# systemctl start activemq
```

- Verify that the broker is listening on 61614 using lsof:

```
[root@stand ~]# lsof -i :61614
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
java 7404 activemq 122 uIPv6 54270 0t0 TCP *:61614
(LISTEN)
```

Configuring nodes to use ActiveMQ

Now we need to create a module to install mcollective on every node and have the nodes' mcollective configuration point back to our message broker. Each node will use a shared key, which we will now generate and sign on our Puppet master as shown here:

```
[root@stand ~]# puppet certificate generate mcollective-servers --ca-location local
Log.newmessage notice 2015-11-19 15:53:16 -0500 mcollective-servers has a
waiting certificate request
Notice: mcollective-servers has a waiting certificate request
true
[root@stand ~]# puppet cert sign mcollective-servers
Log.newmessage notice 2015-11-19 15:53:29 -0500 Signed certificate
request for mcollective-servers
Notice: Signed certificate request for mcollective-servers
Log.newmessage notice 2015-11-19 15:53:29 -0500 Removing file Puppet::SS
L::CertificateRequestmcollective-servers at '/etc/puppetlabs/puppet/ssl/
ca/requests/mcollective-servers.pem'
Notice: Removing file Puppet::SSL::CertificateRequestmcollective-servers
at '/etc/puppetlabs/puppet/ssl/ca/requests/mcollective-servers.pem'
```

We'll now copy the certificate and private keys for this new certificate into our modules files directory and add these files to our module definition. The certificate will be in /etc/puppetlabs/puppet/ssl/ca/signed/mcollective-servers.pem and the private key will be in /etc/puppetlabs/puppet/ssl/private_keys/mcollective-servers.pem. The definitions for these files will be as shown in the following snippet:

```
file {'mcollective_server_cert':
  path    => '/etc/mcollective/ssl/mcollective_public.pem',
  owner   => 0,
  group   => 0,
  mode    => 0640,
  source  => 'puppet:///modules/example/mcollective/mcollective_public.
  pem',
}
file {'mcollective_server_private':
  path    => '/etc/mcollective/ssl/mcollective_private.pem',
  owner   => 0,
  group   => 0,
```

```
    mode      => 0600,
    source   => 'puppet:///modules/example/mcollective/mcollective_
private.pem',
}
```

With the certificates in place, we'll move on to the configuration of the service, as shown in the following snippet:

```
class example::mcollective {
  $mcollective_server = 'puppet.example.com'
  package {'mcollective':
    ensure => true,
  }
  service {'mcollective':
    ensure  => true,
    enable  => true,
    require => [Package['mcollective'],File['mcollective_server_
config']]
  }
  file {'mcollective_server_config':
    path      => '/etc/mcollective/server.cfg',
    owner     => 0,
    group    => 0,
    mode      => 0640,
    content   => template('example/mcollective/server.cfg.erb'),
    require   => Package['mcollective'],
    notify    => Service['mcollective'],
  }
}
```

This is a pretty clean package-file-service relationship. We need to define the mcollective server.cfg configuration file. We'll define this with a template as shown in the following code:

```
main_collective = mcollective
collectives = mcollective
libdir = /usr/libexec/mcollective
daemonize = 1

# logging
logger_type = file
logfile = /var/log/mcollective.log
loglevel = info
logfile = /var/log/mcollective.log
logfacility = user
```

```
keeplogs = 5
max_log_size = 2097152

# activemq
connector = activemq
plugin.activemq.pool.size = 1
plugin.activemq.pool.1.host = <%= mcollective_server %>
plugin.activemq.pool.1.port = 61614
plugin.activemq.pool.1.user = mcollective
plugin.activemq.pool.1.password = PacktPubSecret
plugin.activemq.pool.1.ssl = 1
plugin.activemq.pool.1.ssl.ca = /var/lib/puppet/ssl/certs/ca.pem
plugin.activemq.pool.1.ssl.cert = /var/lib/puppet/ssl/certs/<%= @fqdn %>.pem
plugin.activemq.pool.1.ssl.key = /var/lib/puppet/ssl/private_keys/<%= @fqdn %>.pem
plugin.activemq.pool.1.ssl.fallback = 0

# SSL security plugin settings:
securityprovider = ssl
plugin.ssl_client_cert_dir = /etc/mcollective/ssl/clients
plugin.ssl_server_private = /etc/mcollective/ssl/mcollective_private.pem
plugin.ssl_server_public = /etc/mcollective/ssl/mcollective_public.pem

# Facts, identity, and classes:
identity = <%= @fqdn %>
factsource = yaml
plugin.yaml = /etc/mcollective/facts.yaml
classesfile = /var/lib/puppet/state/classes.txt

registerinterval = 600
```

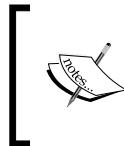
The next thing we need is a populated `facts.yaml` file, as shown in the following snippet, so that we can query facts on the nodes and filter results:

```
file {'facts.yaml':
  path      => '/etc/mcollective/facts.yaml',
  owner     => 0,
  group    => 0,
  mode      => 0640,
  loglevel  => debug,
```

```

content  =>inline_template("----\n<% scope.to_hash.reject { |k,v|  
k.to_s =~ /(uptime_seconds|timestamp|free)/ }.sort.each do |k, v|  
%><%= k %>: <%= v %>\n<% end %>\n"),
require  => Package['mcollective'],
}

```



In the previous example, the `inline_template` uses a call to sort due to random ordering in the hash. Without the sort, the resulting `facts.yaml` file is completely different on each Puppet run, resulting in the entire file being rewritten every time.

Now we're almost there; we have all our nodes pointing to our ActiveMQ server. We need to configure a client to connect to the server.

Connecting a client to ActiveMQ

Clients would normally be installed on the admin user's desktop. We will use `puppet certificate generate` here just as we have in previous examples. We will now outline the steps needed to have a new client connect to mcollective:

1. Create certificates for Thomas and name his certificates thomas:

```
[thomas@client ~]$ puppet certificate generate --ssldir  
~/.mcollective.d/credentials/ --ca-location remote --ca_server  
puppet.example.com --certname thomas thomas
```

2. Sign the cert on `puppet.example.com` (our SSL master):

```
[root@stand ~]# puppet cert sign thomas  
  
Log.newmessage notice 2015-11-21 00:50:41 -0500 Signed certificate  
request for thomas  
  
Notice: Signed certificate request for thomas  
  
Log.newmessage notice 2015-11-21 00:50:41 -0500 Removing file Pupp  
et::SSL::CertificateRequestthomas at '/etc/puppetlabs/puppet/ssl/  
ca/requests/thomas.pem'  
  
Notice: Removing file Puppet::SSL::CertificateRequestthomas at '/  
etc/puppetlabs/puppet/ssl/ca/requests/thomas.pem'
```

3. Retrieve the signed certificate:

```
[root@stand ~]# puppet certificate find thomas --ca-location
remote --ca_server puppet.example.com
-----BEGIN CERTIFICATE-----
MIIFcTCCA1mgAwIBAgIBGjANBgkqhkiG9w0BAQsFADAOoMSYwJAYDVQQDDB1QdXBw
...
-----END CERTIFICATE-----
```

4. Copy this certificate to: `~/.mcollective.d/credentials/certs/thomas.pem`.
5. Download the mcollective-servers key:

```
[root@stand ~]# puppet certificate find mcollective-servers --ca-
location remote --ca_server puppet.example.com
-----BEGIN CERTIFICATE-----
MIIFWzCCA0OgAwIBAgIBEzANBgkqhkiG9w0BAQsFADAOoMSYwJAYDVQQDDB1QdXBw
...
Vd5M01fdYSDKOAb1AXXoMaAn9n9j7AyBhQhie52Og==
-----END CERTIFICATE-----
```

Move this into `~/.mcollective.d/credentials/certs/mcollective-servers.pem`.

6. Download our main CA for certificate verification purposes using the following command:

```
[root@stand ~]# puppet certificate find ca --ca-location remote
--ca_server puppet.example.com
-----BEGIN CERTIFICATE-----
MIIffjCCA2agAwIBAgIBATANBgkqhkiG9w0BAQsFADAOoMSYwJAYDVQQDDB1QdXBw
...
XO+dqA5aAhUUMg==
-----END CERTIFICATE-----
```

Move this into `~/.mcollective.d/credentials/certs/ca.pem`.

7. Now we need to create the configuration file of mco at `~/.mcollective`:

```
connector = activemq
direct_addressing = 1
# ActiveMQ connector settings:
plugin.activemq.pool.size = 1
```

```
plugin.activemq.pool.1.host = puppet.example.com
plugin.activemq.pool.1.port = 61614
plugin.activemq.pool.1.user = mcollective
plugin.activemq.pool.1.password = PacktPubSecret
plugin.activemq.pool.1.ssl = 1
plugin.activemq.pool.1.ssl.ca = /home/thomas/.mcollective.d/
credentials/certs/ca.pem
plugin.activemq.pool.1.ssl.cert = /home/thomas/.mcollective.d/
credentials/certs/thomas.pem
plugin.activemq.pool.1.ssl.key = /home/thomas/.mcollective.d/
credentials/private_keys/thomas.pem
plugin.activemq.pool.1.ssl.fallback = 0
securityprovider = ssl
plugin.ssl_server_public = /home/thomas/.mcollective.d/
credentials/certs/mcollective-servers.pem
plugin.ssl_client_private = /home/thomas/.mcollective.d/
credentials/private_keys/thomas.pem
plugin.ssl_client_public = /home/thomas/.mcollective.d/
credentials/certs/thomas.pem
default_discovery_method = mc
direct_addressing_threshold = 10
ttl = 60
color = 1
rpclimitmethod = first
libdir = /usr/libexec/mcollective
logger_type = console
loglevel = warn
main_collective = mcollective
```

8. Now, we need to add our public key to all the nodes so that they will accept our signed messages. We do this by copying our public key into `example/files/mcollective/clients` and creating a file resource to manage that directory with `recurse => true`, as shown in the following snippet:

```
file {'mcollective_clients':
  ensure  => 'directory',
  path    => '/etc/mcollective/ssl/clients',
  mode    => '0700',
  owner   => 0,
  group   => 0,
  recurse => true,
  source  => 'puppet:///modules/example/mcollective/clients',
}
```

Using mcollective

With everything in place, our client will now pass messages that will be accepted by the nodes, and we in turn will accept the messages signed by the `mcollective-servers` key:

```
[thomas@client ~]$ mco find -v
Discovering hosts using the mc method for 2 second(s) .... 2

client.example.com
puppet.example.com

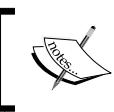
Discovered 2 nodes in 2.06 seconds using the mc discovery plugin
```

Any admin that you wish to add to your team will need to generate a certificate for themselves and have the Puppet CA sign the key. Then they can copy your `.mcollective` file and change the keys to their own. After adding their public key to the `example/mcollective/clients` directory, the nodes will start to accept their messages. You can also add a key for scripts to use; in those cases, using the hostname of the machine, running the scripts will make it easier to distinguish the host that is running the `mco` queries.

Now that `mco` is finally configured, we can use it to generate reports as shown here. The inventory service is a good place to start.

```
[thomas@client ~]$mco inventory client.example.com
Inventory for client.example.com:

  Server Statistics:
    Version: 2.8.6
    Start Time: 2015-11-20 23:12:13 -0800
  Config File: /etc/puppetlabs/mcollective/server.cfg
    Collectives: mcollective
    Main Collective: mcollective
    Process ID: 13665
    Total Messages: 2
  Messages Passed Filters: 2
  Messages Filtered: 0
  Expired Messages: 0
    Replies Sent: 1
  Total Processor Time: 0.21 seconds
    System Time: 0.01 seconds
```



The facts returned in the `inventory` command, and in fact, in any `mco` command, are the redacted facts from the `/etc/puppetlabs/mcollective/facts.yaml` file we created.



Other common uses of `mco` are to find nodes that have classes applied to them, as shown here:

```
[thomas@client ~]$ mco find --wc webserver
www.example.com
```

Another use of `mco` is to find nodes that have a certain value for a fact. You can use regular expression matching using the `/something/` notation, as shown here:

```
[thomas@client ~]$ mco find --wf hostname=/^node/
node2.example.com
node1.example.com
```

Using the built-in modules, it's possible to start and stop services. Check file contents and write your own modules to perform tasks.

Ansible

When you need to orchestrate changes across a large number of servers, some of which may not currently be functioning, `mcollective` is a very good tool. When running Puppet in a large organization there are several tasks that need to be performed in an orchestrated fashion with a small number of machines. In my opinion, Ansible is a great tool for these small changes across multiple machines. I've used Ansible through Git hook scripts to deploy updated code across a set of Puppet master machines. More information on Ansible can be found at <http://docs.ansible.com/>.

Summary

Reports help you understand when things go wrong. Using some of the built-in report types, it's possible to alert your admins to Puppet failures. The GUIs mentioned here allow you to review Puppet run logs. Foreman has the most polished feel and makes it easier to link directly to reports and search for reports. `mcollective` is an orchestration utility that allows you to actively query and modify all the nodes in an organized manner interactively via a message broker.

In the next chapter, we will be installing PuppetDB and creating exported resources.

8

Exported Resources

When automating tasks among many servers, information from one node may affect the configuration of another node or nodes. For example, if you configure DNS servers using Puppet, then you can have Puppet tell the rest of your nodes where all the DNS servers are located. This sharing of information is called **catalog storage and searching** in Puppet.

Catalog storage and searching was previously known as **storeconfigs** and enabled using the `storeconfig` option in `puppet.conf`. Storeconfigs was able to use SQLite, MySQL, and PostgreSQL; it is now deprecated in favor of **PuppetDB**.

The current method of supporting exported resources is PuppetDB, which uses Java and PostgreSQL and can support hundreds to thousands of nodes with a single PuppetDB instance. Most scaling issues with PuppetDB can be solved by beefing up the PostgreSQL server, either adding a faster disk or more CPU, depending on the bottleneck.

We will begin our discussion of exported resources by configuring PuppetDB. We will then discuss exported resource concepts and some example usage.

Configuring PuppetDB – using the Forge module

The easy way to configure PuppetDB is to use the `puppetdb` Puppet module on Puppet Forge at <https://forge.puppetlabs.com/puppetlabs/puppetdb>. We will install PuppetDB using the module first to show how quickly you can deploy PuppetDB. In the subsequent section, we'll configure PuppetDB manually to show how all the components fit together.

The steps to install and use PuppetDB that we will outline are as follows:

1. Install the `puppetdb` module on Puppet master (`stand`).
2. Install `puppetlabs-repo` and Puppet on PuppetDB host.
3. Deploy the `puppetdb` module onto PuppetDB host.
4. Update the configuration of the Puppet master to use PuppetDB.

We will start with a vanilla EL6 machine and install PuppetDB using the `puppetdb` module. In *Chapter 4, Public Modules*, we used a `Puppetfile` in combination with `librarian-puppet` or `r10k` to download modules. We used the `puppetdb` module since it was a good example of dependencies; we will rely on PuppetDB being available to our catalog worker for this example. If you do not already have PuppetDB downloaded, do it using one of those methods or simply use `puppet module install puppetlabs-puppetdb` as shown in the following screenshot:

```
[root@stand modules]# puppet module install puppetlabs-puppetdb
Notice: Preparing to install into /etc/puppetlabs/code/environments/production/modules...
Notice: Downloading from https://forgeapi.puppetlabs.com...
Notice: Installing -- do not interrupt ...
/etc/puppetlabs/code/environments/production/modules
└── puppetlabs-puppetdb (v5.0.0)
    ├── puppetlabs-firewall (v1.7.1)
    ├── puppetlabs-inifile (v1.4.2)
    └── puppetlabs-postgresql (v4.6.0)
        ├── puppetlabs-apt (v2.2.0)
        ├── puppetlabs-concat (v1.2.4)
        └── puppetlabs-stdlib (v4.9.0)
[root@stand modules]#
```

After installing the `puppetdb` module, we need to install the `puppetlabs` repo on our PuppetDB machine and install Puppet using the following command:

```
[root@puppetdb ~]# yum -y install http://yum.puppetlabs.com/puppetlabs-release-pcl-el-7.noarch.rpm
puppetlabs-release-pcl-el-7.noarch.rpm | 5.1 kB
00:00:00
Examining /var/tmp/yum-root-dfBZAN/puppetlabs-release-pcl-el-7.noarch.rpm: puppetlabs-release-pcl-1.0.0-1.el6.noarch
Marking /var/tmp/yum-root-dfBZAN/puppetlabs-release-pcl-el-7.noarch.rpm as an update to puppetlabs-release-pcl-0.9.2-1.el7.noarch
Resolving Dependencies
--> Running transaction check
...
root@puppetdb ~]# yum -y install puppetdb
```

```
Resolving Dependencies
--> Running transaction check
--> Package puppetdb.noarch 0:3.2.0-1.el7 will be installed
--> Processing Dependency: net-tools for package: puppetdb-3.2.0-1.el7.noarch
--> Processing Dependency: java-1.8.0-openjdk-headless for package: puppetdb-3.2.0-1.el7.noarch
--> Running transaction check
--> Package java-1.8.0-openjdk-headless.x86_64 1:1.8.0.65-2.b17.el7_1 will be installed
--> Processing Dependency: jpackage-utils for package: 1:java-1.8.0-openjdk-headless-1.8.0.65-2.b17.el7_1.x86_64
...

```

Our next step is to deploy PuppetDB on the PuppetDB machine using Puppet. We'll create a wrapper class to install and configure PuppetDB on our master, as shown in the following code (in the next chapter this will become a profile). Wrapper classes, or profiles, are classes that bundle lower-level classes (building blocks) into higher-level classes.

```
classpdb {
  # puppetdb class
  class { 'puppetdb::server': }
  class { 'puppetdb::database::postgresql': listen_addresses => '*' }
}
```

At this point, the PuppetDB server also needs network ports opened in iptables; the two ports are 5432 (postgresql) and 8081 (puppetdb). Using our knowledge of the `firewall` module, we could do this with the following snippet included in our `pdb` class:

```
firewall {'5432 postgresql':
  action => 'accept',
  proto  => 'tcp',
  dport  => '5432',
}
firewall {'8081 puppetdb':
  action => 'accept',
  proto  => 'tcp',
  dport  => '8081',
}
```

Exported Resources

We then apply this `pdb` class to our PuppetDB machine. For this example, I used the `hiera_include` method and the following `puppetdb.yaml` file:

```
---  
classes: pdb
```

Now we run Puppet agent on PuppetDB to have PuppetDB installed (running Puppet agent creates the SSL keys for our PuppetDB server as well; remember to sign those on the master).

Back on our workers, we need to tell Puppet to use PuppetDB; we can do this by defining a `puppet::master` class that configures Puppet and applying it to our workers:

```
class puppet::master {  
    class {'puppetdb::master::config':  
        puppetdb_server      => 'puppetdb.example.com',  
        puppet_service_name  => 'httpd',  
    }  
}
```

Now we configure our `stand.yaml` file to include the previous class as follows:

```
---  
classes: puppet::master
```

The Puppet master will need to be able to resolve `puppetdb.example.com`, either through DNS or static entries in `/etc/hosts`. Now run Puppet on our Puppet master to have `puppetserver` configured to use PuppetDB. The master will attempt to communicate with the PuppetDB machine over port 8081. You'll need the firewall (`iptables`) rules to allow this access at this point.

Now we can test that PuppetDB is operating by using the `puppet node status` command as follows:

```
[root@stand ~]# puppet node status puppetdb.example.com
```

```
Currently active  
Last catalog: 2015-11-27T10:43:42.243Z  
Last facts: 2015-11-27T10:43:26.539Z
```

Manually installing PuppetDB

The `puppetlabs/puppetdb` module does a great job of installing PuppetDB and getting you running quickly. Unfortunately, it also obscures a lot of the configuration details. In the enterprise, you'll need to know how all the parts fit together. We will now install PuppetDB manually using the following five steps:

1. Install Puppet and PuppetDB.
2. Install and configure PostgreSQL.
3. Configure PuppetDB to use PostgreSQL.
4. Start PuppetDB and open firewall ports.
5. Configure the Puppet master to use PuppetDB.

Installing Puppet and PuppetDB

To manually install PuppetDB, start with a fresh machine and install the `puppetlabs-pc1` repository, as in previous examples. We'll call this new server `puppetdb-manual.example.com` to differentiate it from our automatically installed PuppetDB instance (`puppetdb.example.com`).

Install Puppet, do a Puppet agent run using the following command to generate certificates, and sign them on the master as we did when we used the `puppetlabs/puppetdb` module. Alternatively, use `puppet certificate generate` as we did in previous chapters:

```
[root@puppetdb-manual ~]# yum -y install http://yum.puppetlabs.com/
puppetlabs-release-pc1-el-6.noarch.rpm
[root@puppetdb-manual ~]# yum install puppet-agent
[root@puppetdb-manual ~]# puppet agent -t
```

Sign the certificate on the master as follows:

```
[root@stand ~]# puppet cert list
"puppetdb-manual.example.com" (SHA256) 90:5E:9B:D5:28:50:E0:43:82:F4:F5
:D9:21:0D:C3:82:1B:7F:4D:BB:DC:C0:E5:ED:A1:EB:24:85:3C:01:F4:AC
[root@stand ~]# puppet cert sign puppetdb-manual.example.com
Notice: Signed certificate request for puppetdb-manual.example.com
Notice: Removing file Puppet::SSL::CertificateRequest 'puppetdb-manual.
example.com' at '/etc/puppetlabs/puppet/ssl/ca/requests/puppetdb-manual.
example.com.pem'
```

Back on `puppetdb-manual`, install `puppetdb` as follows:

```
[root@puppetdb-manual ~]# yum -q -y install puppetdb
```

Installing and configuring PostgreSQL

If you already have an enterprise PostgreSQL server configured, you can simply point PuppetDB at that instance. PuppetDB 3.2 only supports PostgreSQL versions 9.4 and higher. To install PostgreSQL, install the `postgresql-server` package and initialize the database as follows:

```
[root@puppetdb-manual ~]# yum install http://yum.postgresql.org/9.4/
redhat/rhel-7-x86_64/pgdg-redhat94-9.4-2.noarch.rpm -q -y
[root@puppetdb-manual ~]# yum -q -y install postgresql94-server
[root@puppetdb-manual ~]# postgresql-setup initdb
Initializing database ... OK
[root@puppetdb-manual ~]# systemctl start postgresql-9.4
```

Next create the `puppetdb` database (allowing the `puppetdb` user to access that database) as follows:

```
[root@puppetdb-manual ~]# sudo -iu postgres
$ createuser -DRSP puppetdb
Enter password for new role: PacktPub
Enter it again: PacktPub
$ createdb -E UTF8 -O puppetdb puppetdb
```

Allow PuppetDB to connect to the PostgreSQL server using md5 on the localhost since we'll keep PuppetDB and the PostgreSQL server on the same machine (`puppetdb-manual.example.com`).



You will need to change the allowed address rules from `127.0.0.1/32` to that of the PuppetDB server if PuppetDB is on a different server than the PostgreSQL server.

Edit `/var/lib/pgsql/9.4/data/pg_hba.conf` and add the following:

```
local  puppetdb  puppetdb      md5
host   puppetdb  puppetdb      127.0.0.1/32  md5
host   puppetdb  puppetdb      ::1/128       md5
```



The default configuration uses ident authentication; you must remove the following lines:

```
local  all        all          ident
host   all        all          127.0.0.1/32  ident
host   all        all          ::1/128       ident
```

Restart PostgreSQL and test connectivity as follows:

```
[root@puppetdb-manual ~]# systemctl restart postgresql-9.4
[root@puppetdb-manual ~]# psql -h localhost puppetdb puppetdb
Password for user puppetdb: PacktPub
psql (9.4.5)
Type "help" for help.

puppetdb=> \d
No relations found.
puppetdb=> \q
```

Now that we've verified that PostgreSQL is working, we need to configure PuppetDB to use PostgreSQL.

Configuring puppetdb to use PostgreSQL

Locate the database .ini file in /etc/puppetlabs/puppetdb/conf.d and replace it with the following code snippet:

```
[database]
classname = org.postgresql.Driver
subprotocol = postgresql
subname = //localhost:5432/puppetdb
username = puppetdb
password = PacktPub
```

If it's not present in your file, configure automatic tasks of PuppetDB such as garbage collection (gc-interval), as shown in the following code. PuppetDB will remove stale nodes every 60 minutes. For more information on the other settings, refer to the Puppet Labs documentation at <http://docs.puppetlabs.com/puppetdb/latest/configure.html>:

```
gc-interval = 60
log-slow-statements = 10
report-ttl = 14d
syntax_pgs = true
conn-keep-alive = 45
node-ttl = 0s
conn-lifetime = 0
node-purge-ttl = 0s
conn-max-age = 60
```

Start PuppetDB using the following command:

```
[root@puppetdb_manual ~]# systemctl start puppetdb
```

Configuring Puppet to use PuppetDB

Perform the following steps to configure Puppet to use PuppetDB.

To use PuppetDB, the worker will need the `puppetdb terminus` package; we'll install that first by using the following command:

```
# yum -y install puppetdb-termini
```

Create `/etc/puppetlabs/puppet/puppetdb.conf` and point PuppetDB at `puppetdb-manual.example.com`:

```
[main]
server_urls = https://puppetdb-manual.example.com:8081/
soft_write_failure = false
```

Tell Puppet to use PuppetDB for storeconfigs by adding the following in the `[master]` section of `/etc/puppetlabs/puppet/puppet.conf`:

```
[master]
storeconfigs = true
storeconfigs_backend = puppetdb
```

Next, create a `routes.yaml` file in the `/etc/puppetlabs/puppet` directory that will make Puppet use PuppetDB for inventory purposes:

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

Restart `puppetserver` and verify that PuppetDB is working by running `puppet agent` again on `puppetdb-manual.example.com`. After the second `puppet agent` runs, you can inspect the PostgreSQL database for a new catalog as follows:

```
[root@puppetdb-manual ~]# psql -h localhost puppetdb puppetdb
Password for user puppetdb:
psql (9.4.5)
Type "help" for help.

puppetdb=> \x
Expanded display is on.
puppetdb=> SELECT * from catalogs;
```

```
- [ RECORD 1 ] -----+
id| 1
hash          | \x13980e07b72cf8e02ea247c3954efdc2cdabbbe0
transaction_uuid | 9ce673db-6af2-49c7-b4c1-6eb83980ac57
certname       | puppetdb-manual.example.com
producer_timestamp | 2015-12-04 01:27:19.211-05
api_version    | 1
timestamp       | 2015-12-04 01:27:19.613-05
catalog_version | 1449210436
environment_id  | 1
code_id         |
```

Exported resource concepts

Now that we have PuppetDB configured, we can begin exporting resources into PuppetDB. In *Chapter 5, Custom Facts and Modules*, we introduced virtual resources. Virtual resources are resources that are defined but not instantiated. The concept with virtual resources is that a node has several resources defined, but only one or a few resources are instantiated. Instantiated resources are not used in catalog compilation. This is one method of overcoming some "duplicate definition" type problems. The concept with exported resources is quite similar; the difference is that exported resources are published to PuppetDB and made available to any node in the enterprise. In this way, resources defined on one node can be instantiated (realized) on another node.

What actually happens is quite simple. Exported resources are put into the `catalog_resources` table in the PostgreSQL backend of PuppetDB. The table contains a column named `exported`. This column is set to `true` for exported resources. When trying to understand exported resources, just remember that exported resources are just entries in a database.

To illustrate exported resources, we will walk through a few simple examples. Before we start, you need to know two terms used with exported resources: declaring and collecting.

Declaring exported resources

Exported resources are declared with the @@ operator. You define the resource as you normally would, but prepend the definition with @@. For example, consider the following host resource:

```
host {'exported':
  host_aliases => 'exported-resources',
  ip      => '1.1.1.1',
}
```

It can be declared as an exported resource as follows:

```
@@host {'exported':
  host_aliases => 'exported-resources',
  ip      => '1.1.1.1',
}
```

Any resource can be declared as an exported resource. The process of realizing exported resources is known as collecting.

Collecting exported resources

Collecting is performed using a special form of the collecting syntax. When we collected virtual resources, we used < | | > to collect the resources. For exported resources, we use << | | >>. To collect the previous host resource, we use the following:

```
Host <<| |>>
```

To take advantage of exported resources, we need to think about what we are trying to accomplish. We'll start with a simplified example.

Simple example – a host entry

It makes sense to have static host entries in /etc/hosts for some nodes, since DNS outages may disrupt the services provided by those nodes. Examples of such services are backups, authentication, and Kerberos. We'll use LDAP (authentication) in this example. In this scenario, we'll apply the ldap::server class to any LDAP server and add a collector for Host entries to our base class (the base class will be a default applied to all nodes). First, declare the exported resource in ldap::server, as shown in the following code snippet:

```
class ldap::server {
  @@host {"ldap-$::hostname":
    host_aliases => ["$::fqdn", 'ldap'],
```

```

    ip          => "$::ipaddress",
}
}

```

This will create an exported entry on any host to which we apply the `ldap::server` class. We'll apply this class to `node2` and then run Puppet to have the resource exported. After running Puppet agent on `ldapserver1`, we will examine the contents of PuppetDB, as shown in the following screenshot:

```

puppetdb=> \x on
Expanded display is on.
puppetdb=> SELECT * FROM catalog_resources WHERE exported=TRUE and title like '%ldapserver1%';
-[ RECORD 1 ]-----
catalog_id | 2
resource   | \x04e564309958aaa40d27c6dbd7770f706c5c8ef6
tags       | {server,ldap,host,class,ldap::server,default,node,ldap-ldapserver1}
type       | Host
title     | ldap-ldapserver1
exported   | t
file      | /etc/puppetlabs/code/environments/production/modules/ldap/manifests/server.pp
line      | 2
puppetdb=>

```

The `catalog_resources` table holds the catalog resource mapping information. Using the resource ID from this table, we can retrieve the contents of the resource from the `resource_params` table, as shown in the following screenshot:

```

puppetdb=> SELECT * FROM resource_params WHERE resource='\x04e564309958aaa40d27c6dbd7770f706c5c8ef6';
-[ RECORD 1 ]-----
resource | \x04e564309958aaa40d27c6dbd7770f706c5c8ef6
name     | host_aliases
value    | ["ldapserver1.example.com", "ldap"]
-[ RECORD 2 ]-----
resource | \x04e564309958aaa40d27c6dbd7770f706c5c8ef6
name     | ip
value    | "10.0.2.15"
puppetdb=>

```

As we can see, the `ldapserver1` host entry has been made available in PuppetDB. The `host_aliases` and `ip` information has been stored in PuppetDB.

To use this exported resource, we will need to add a collector to our `base` class as follows:

```

class base {
  Host <<| |>>
}

```

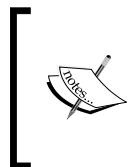
Now, when we run `puppet agent` on any host in our network (any host that has the base class applied), we will see the following host entry:

```
[root@client ~]# grep ldap /etc/hosts  
10.0.2.15 ldap-ldapserver1 ldapserver1.example.comldap
```

The problem with this example is that every host with `ldap::server` applied will be sent to every node in the enterprise. To make things worse, any exported host resource will be picked up by our collector. We need a method to be specific when collecting our resources. Puppet provides tags for this purpose.

Resource tags

Resource tags are **metaparameters** available to all resources in Puppet. They are used in collecting only and do not affect the definition of resources.



Metaparameters are part of how Puppet compiles the catalog and not part of the resource to which they are attached. Metaparameters include `before`, `notify`, `require`, and `subscribe`. More information on metaparameters is available at <http://docs.puppetlabs.com/references/latest/metaparameter.html>.



Any tags explicitly set on a resource will be appended to the array of tags. In our previous example, we saw the tags for our host entry in the PostgreSQL output as follows, but we didn't address what the tags meant:

```
{server,ldap,host,class,ldap::server,default,node,ldap-ldapserver1}
```

All these tags are defaults set by Puppet. To illustrate how tags are used, we can create multiple exported host entries with different tags. We'll start with adding a tag search to our Host collector in the base class as follows:

```
Host <<| tag == 'ldap-server' |>>
```

Then we'll add an `ldap-client` exported host resource to the base class with the tag '`ldap-client`' as follows:

```
@@host {"ldap-client-$::hostname":  
    host_aliases => ["$::fqdn", "another-$::hostname"],  
    ip          => "$::ipaddress",  
    tag         => 'ldap-client',  
}
```

Now all nodes will only collect Host resources marked as `ldap-server`. Every node will create an `ldap-client` exported host resource; we'll add a collector for those to the `ldap::server` class:

```
Host <<| tag == 'ldap-client' |>>
```

One last change: we need to make our `ldap-server` resource-specific, so we'll add a tag to it in `ldap::server` as follows:

```
@@host {"ldap-$::hostname":
  host_aliases => ["$::fqdn", 'ldap'],
  ip           => "$::ipaddress",
  tag          => 'ldap-server',
}
```

Now every node with the `ldap::server` class exports a host resource tagged with `ldap-server` and collects all host resources tagged with `ldap-client`. After running Puppet on master and client nodes 1 and 2, we see the following on our `ldapserver1` as the host resources tagged with `ldap-client` get defined:

```
[root@ldapserver1 ~]# puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Loading facts
Info: Caching catalog for ldapserver1.example.com
Info: Applying configuration version '1449333905'
Notice: /Stage[main]/Base/Host[ldap-client-ldapserver1]/ensure: created
Info: Computing checksum on file /etc/hosts
Notice: /Stage[main]/Ldap::Server/Host[ldap-client-stand]/ensure: created
Notice: /Stage[main]/Ldap::Server/Host[ldap-client-client]/ensure: created
Notice: Applied catalog in 0.09 seconds
[root@ldapserver1 ~]# grep ldap /etc/hosts
10.0.2.15      ldap-ldapserver1      ldapserver1.example.com ldap
10.0.2.15      ldap-client-ldapserver1  ldapserver1.example.com another-ldapserver1
10.0.2.15      ldap-client-stand    puppet.example.com another-stand
10.0.2.15      ldap-client-client   client.example.com another-client
[root@ldapserver1 ~]#
```

Exported SSH keys

Most exported resource documentation starts with an SSH key example. `sshkey` is a Puppet type that creates or destroys entries in the `ssh_known_hosts` file used by SSH to verify the validity of remote servers. The `sshkey` example is a great use of exported resources, but since most examples put the declaration and collecting phases in the same class, it may be a confusing example for those starting out learning exported resources. It's important to remember that exporting and collecting are different operations.

sshkey collection for laptops

We'll outline an enterprise application of the `sshkey` example and define a class for login servers—any server that allows users to log in directly. Using that class to define exported resources for `ssh_host_keys`, we'll then create an `ssh_client` class that collects all the login server `ssh_keys`. In this way, we can apply the `ssh_client` class to any laptops that might connect and have them get updated SSH host keys. To make this an interesting example, we'll run Puppet as non-root on the laptop and have Puppet update the user's `known_hosts` file `~/.ssh/known_hosts` instead of the system file. This is a slightly novel approach to running Puppet without root privileges.

We'll begin by defining an `example::login_server` class that exports the RSA and DSA SSH host keys. RSA and DSA are the two types of encryption keys that can be used by the SSH daemon; the name refers to the encryption algorithm used by each key type. We will need to check if a key of each type is defined as it is only a requirement that one type of key be defined for the SSH server to function, as shown in the following code:

```
class example::login_server {
  if ( $::sshrsakey != undef ) {
    @@sshkey { "$::fqdn-rsa":
      host_aliases => ["$::hostname", "$::ipaddress"],
      key          => "$::sshrsakey",
      type         => 'rsa',
      tag          => 'example::login_server',
    }
  }
  if ( $::sshdsakey != undef ) {
    @@sshkey { "$::fqdn-dsa":
      host_aliases => ["$::hostname", "$::ipaddress"],
      key          => "$::sshdsakey",
      type         => 'dsa',
      tag          => 'example::login_server',
    }
  }
}
```

This class will export two SSH key entries, one for the `rsa` key and another for the `dsa` key. It's important to populate the `host_aliases` array as we have done so that both the IP address and short hostname are verified with the key when using SSH.

Now we could define an `example::laptop` class that simply collects the keys and applies them to the system-wide `ssh_known_hosts` file. Instead, we will define a new fact, `homedir` in `base/lib/facter/homedir.rb`, to determine if Puppet is being run by a non-root user, as follows:

```
Facter.add(:homedir) do
  if Process.uid != 0 and ENV['HOME'] != nil
    setcode do
      begin
        ENV['HOME']
      rescue LoadError
        nil
      end
    end
  end
end
```

This simple fact checks the UID of the running Puppet process; if it is not 0 (root), it looks for the environment variable `HOME` and sets the fact `homedir` equal to the value of that environment variable.

Now we can key off this fact as a top scope variable in our definition of the `example::laptop` class as follows:

```
class example::laptop {
  # collect all the ssh keys
  if $::homedir != undef {
    Sshkey<<| tag == 'login_server' |>> {
      target => "$::homedir/.ssh/known_hosts"
    }
  } else {
    Sshkey<<| tag == 'login_server' |>>
  }
}
```

Depending on the value of the `$::homedir` fact, we either define system-wide SSH keys or userdir keys. The SSH key collector (`Sshkey<<| tag == 'login_server' |>>`) uses the tag `login_server` to restrict the SSH key resources to those defined by our `example::login_server` class.

To test this module, we apply the `example::login_server` class to two servers, `ssh1` and `ssh2`, thereby creating the exported resources. Now on our laptop, we run Puppet as ourselves and sign the key on Puppet master.



If Puppet has already run as root or another user, the certificate may have already been generated for your laptop hostname; use the `--certname` option to `puppet agent` to request a new key.

We add the `example::laptop` class to our laptop machine and examine the output of our Puppet run.

Our laptop is likely not a normal client of our Puppet master, so when calling Puppet agent, we define the `puppetserver` and environment as follows:

```
t@mylaptop ~ $ puppet agent -t --environment production --server puppet.example.com --waitforcert 60
Info: Creating a new SSL key for mylaptop.example.com
Info: Caching certificate for ca
Info: csr_attributes file loading from /home/thomas/.puppetlabs/etc/puppet/csr_attributes.yaml
Info: Creating a new SSLcertificate request for mylaptop.example.com
Info: Certificate Request fingerprint (SHA256): 97:86:BF:BD:79:FB:B2:AC:0C:8E:80:D0:5E:D0:18:F9:42:BD:25:CC:A9:25:44:7B:30:7B:F9:C6:A2:11:6E:61
Info: Caching certificate for ca
Info: Caching certificate for mylaptop.example.com
Info: Caching certificate_revocation_list for ca
Info: Retrieving pluginfacts
...
Info: Loading facts
Info: Caching catalog for mylaptop.example.com
Info: Applying configuration version '1449337295'
Notice: /Stage[main]/Example::Laptop/Sshkey[ssh1.example.com-rsa]/ensure: created
Info: Computing checksum on file /home/thomas/.ssh/known_hosts
Notice: /Stage[main]/Example::Laptop/Sshkey[ssh2.example.com-rsa]/ensure: created
Info: Stage[main]: Unscheduling all events on Stage[main]
Notice: Applied catalog in 0.12 seconds
```

Since we ran the agent as non-root, the system-wide SSH keys in `ssh_known_hosts` cannot have been modified. Looking at `~/.ssh/known_hosts`, we see the new entries at the bottom of the file as follows:

```
ssh1.example.com-rsa,ssh1,10.0.2.15ssh-rsaAAAAB3NzaC1yc2...
ssh2.example.com-rsa,ssh2,10.0.2.15ssh-rsaAAAbd3dz56c2E...
```

Putting it all together

Any resource can be exported, including defined types and your own custom types. Tags may be used to limit the set of exported resources collected by a collector. Tags may include local variables, facts, and custom facts. Using exported resources, defined types, and custom facts, it is possible to have Puppet generate complete interactions without intervention (automatically).

As an abstract example, think of any clustered service where members of a cluster need to know about the other members of the cluster. You could define a custom fact, `clusternode`, that defines the name of the cluster based on information either on the node or in a central **Configuration Management Database (CMDB)**.



CMDBs are the data warehouses of an organization. Examples of CMDBs include OneCMDB, Itop, or BMC Atrium.



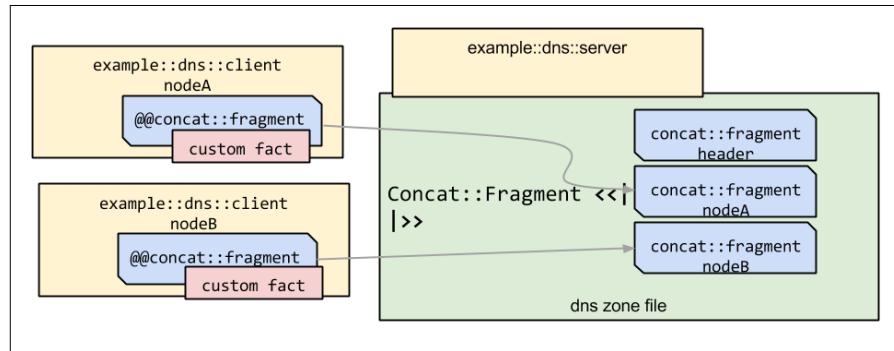
You would then create a cluster module, which would export firewall rules to allow access from each node. The nodes in the cluster would collect all the exported rules based on the relationship `tag=="clusternode"`. Without any interaction, a complex firewall rule relationship would be built up between cluster members. If a new member is added to the cluster, the rules will be exported and, with the next Puppet run, the node will be permitted access to the other cluster members.

Another useful scenario is where there are multiple slave nodes that need to be accessed by a master node, such as with backup software or a software distribution system. The master node needs the slave nodes to allow access to them. The slave nodes need to know which node is the master node. In this relationship, you would define a master and a slave module and apply them accordingly. The slave node would export its host configuration information, and the master would export both its firewall access rule and master configuration information. The master would collect all the slave configuration resources. The slaves would each collect the firewall and configuration information from the master. The great thing about this sort of configuration is that you can easily migrate the master service to a new node. As slaves check into Puppet, they will receive the new master configuration and begin pointing at the new master node.

Exported Resources

To illustrate this concept, we will go through a DNS configuration example. We will configure a DNS server with the `example::dns::server` class. We will then configure clients using a `example::dns::client` class. DNS servers are configured with zone files. Zone files come in two forms: forward zones map hostnames to IP addresses and reverse zones map IP address to hostnames. To make a fully functioning DNS implementation, our clients will export a `concat::fragment` resource, which will be collected on the master and used to build both the forward and reverse DNS zone files.

The following diagram outlines the process where two nodes export `concat::fragment` resources that are assembled with a header into a zone file on the DNS server node:



To start, we will define two custom facts that produce the reverse of the IP address suitable for use in a DNS reverse zone, and the network in **Classless Inter-Domain Routing (CIDR)** notation used to define the reverse zone file, as follows:

```
# reverse.rb
# Set a fact for the reverse lookup of the network
require 'ipaddr'
require 'puppet/util/ipcidr'

# define 2 facts for each interface passed in
def reverse(dev)
  # network of device
  ip = IPAddr.new(Facter.value("network_#{dev}"))
  # network in cidr notation (uuu.vvv.www.xxx/yy)
  nm = Puppet::Util::IPCidr.new(Facter.value("network_#{dev}")).
    mask(Facter.value("netmask_#{dev}"))
  cidr = nm.cidr
```

```

# set fact for network in reverse vvv.www.uuu.in-addr.arpa
Facter.add("reverse_#{dev}") do
  setcode do ip.reverse.to_s[2..-1] end
end

# set fact for network in cidr notation
Facter.add("network_cidr_#{dev}") do
  #
  setcode do cidr end
end
end

```

We put these two fact definitions into a Ruby function so that we can loop through the interfaces on the machine and define the facts for each interface as follows:

```

# loop through the interfaces, defining the two facts for each
interfaces = Facter.value('interfaces').split(',')
interfaces.each do
  |eth| reverse(eth)
end

```

Save this definition in `example/lib/facter/reverse.rb` and then run Puppet to synchronize the fact definition down to the nodes. After the fact definition has been transferred, we can see its output for `dns1` (IP address 192.168.1.54) as follows:

```

[root@dns1 ~]# facter -p interfaces
enp0s3,enp0s8,lo
[root@dns1 ~]# facter -p ipaddress_enp0s8
192.168.1.54
[root@dns1 ~]# facter -p reverse_enp0s8network_cidr_enp0s8
network_cidr_enp0s8 => 192.168.1.0/24
reverse_enp0s8 =>1.168.192.in-addr.arpa

```

In our earlier custom fact example, we built a custom fact for the zone based on the IP address. We could use the fact here to generate zone-specific DNS zone files. To keep this example simple, we will skip this step. With our fact in place, we can export our client's DNS information in the form of `concat::fragments` that can be picked up by our master later. To define the clients, we'll create an `example::dns::client` class as follows:

```

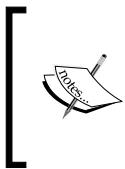
class example::dns::client
(
  String $domain = 'example.com',
  String $search = prod.example.comexample.com'
)

```

Exported Resources

We start by defining the search and domain settings and providing defaults. If we need to override the settings, we can do so from Hiera. These two settings would be defined as the following in a Hiera YAML file:

```
---  
example::dns::client::domain: 'subdomain.example.com'  
example::dns::client::search: 'sub.example.comprod.example.com'
```



Be careful when modifying `/etc/resolv.conf`. This can change the way Puppet defines `certname` used to verify the nodes' identity to the puppetserver. If you change your domain, a new certificate will be requested and you will have to sign the new certificate before you can proceed.



We then define a `concat` container for `/etc/resolv.conf` as follows:

```
concat {'/etc/resolv.conf':  
    mode => '0644',  
}  
  
# search and domain settings  
concat::fragment{'resolv.conf search/domain':  
    target => '/etc/resolv.conf',  
    content => "search $search\nndomain $domain\n",  
    order => 07,  
}
```

The `concat::fragment` will be used to populate the `/etc/resolv.conf` file on the client machines. We then move on to collect the nameserver entries, which we will later export in our `example::dns::server` class using the tag '`resolv.conf`'. We use the tag to make sure we only receive fragments related to `resolv.conf` as follows:

```
Concat::Fragment <<| tag == 'resolv.conf' |>> {  
    target => '/etc/resolv.conf'  
}
```

We use a piece of syntax we haven't used yet for exported resources called **modify on collect**. With `modify` on `collect`, we override settings in the exported resource when we collect. In this case, we are utilizing `modify` on `collect` to modify the exported `concat::fragment` to include a target. When we define the exported resource, we leave the target off so that we do not need to define a `concat` container on the server. We'll be using this same trick when we export our DNS entries to the server.

Next we export our zone file entries as `concat::fragments` and close the class definition as follows:

```
@@concat::fragment {"zone example $::hostname":
  content => "$::hostname A $::ipaddress\n",
  order   => 10,
  tag     => 'zone.example.com',
}
$lastoctet = regsubst($::ipaddress_enp0s8, '^([0-9]+)[.][0-9]+[.][0-9]+[.]([0-9]+)[.]([0-9]+)$', '\4')
@@concat::fragment {"zone reverse $::reverse_enp0s8 $::hostname":
  content => "$lastoctetPTR $::fqdn.\n",
  order   => 10,
  tag     => "reverse.$::reverse_enp0s8",
}
}
```

In the previous code, we used the `regsubst` function to grab the last octet from the nodes' IP address. We could have made another custom fact for this, but the `regsubst` function is sufficient for this usage.

Now we move on to the DNS server to install and configure binds named daemon; we need to configure the `named.conf` file and the zone files. We'll define the `named.conf` file from a template first as follows:

```
class example::dns::server {

  # setup bind
  package {'bind': }
  service {'named': require => Package['bind'] }

  # configure bind
  file {'/etc/named.conf':
    content => template('example/dns/named.conf.erb'),
    owner   => 0,
    group   => 'named',
    require  => Package['bind'],
    notify   => Service['named']
  }
}
```

Exported Resources

Next we'll define an exec that reloads `named` whenever the zone files are altered as follows:

```
exec { 'named reload':
  refreshonly => true,
  command     => 'systemctl reload named',
  path         => '/bin:/sbin',
  require      => Package['bind'],
}
```

At this point, we'll export an entry from the server, defining it as `nameserver` as follows (we already defined the collection of this resource in the client class):

```
@@concat::fragment {"resolv.confnameserver $::hostname":
  content => "nameserver $::ipaddress\n",
  order   => 10,
  tag     => 'resolv.conf',
}
```

Now for the zone files; we'll define concat containers for the forward and reverse zone files and then header fragments for each as follows:

```
concat {'/var/named/zone.example.com':
  mode    => '0644',
  notify  => Exec['named reload'],
}
concat {'/var/named/reverse.122.168.192.in-addr.arpa':
  mode    => '0644',
  notify  => Exec['named reload'],
}
concat::fragment {'zone.example header':
  target  => '/var/named/zone.example.com',
  content => template('example/dns/zone.example.com.erb'),
  order   => 01,
}
concat::fragment {'reverse.122.168.192.in-addr.arpa header':
  target  => '/var/named/reverse.122.168.192.in-addr.arpa',
  content => template('example/dns/reverse.122.168.192.in-addr.arpa.
erb'),
  order   => 01,
}
```

Our clients exported `concat::fragments` for each of the previous zone files. We collect them here and use the same `modify on collect` syntax as we did for the client as follows:

```
Concat::Fragment <<| tag == "zone.example.com" |>> {
    target => '/var/named/zone.example.com'
}
Concat::Fragment <<| tag == "reverse.122.168.192.in-addr.arpa" |>> {
    target => '/var/named/reverse.122.168.192.in-addr.arpa'
}
```

The server class is now defined. We only need to create the template and header files to complete our module. The `named.conf.erb` template makes use of our custom facts as well, as shown in the following code:

```
options {
    listen-on port 53 { 127.0.0.1; <%= @ipaddress_enp0s8 -%>; };
    listen-on-v6 port 53 { ::1; };
    directory    "/var/named";
    dump-file    "/var/named/data/cache_dump.db";
    statistics-file "/var/named/data/named_stats.txt";
    memstatistics-file "/var/named/data/named_mem_stats.txt";
    allow-query   { localhost; <%- @interfaces.split(',').each do |eth| if has_variable?("network_cidr_#{eth}") then -%><%= scope.lookupvar("network_cidr_#{eth}") -%>; <%- end end -%> };
    recursion yes;

    dnssec-enable yes;
    dnssec-validation yes;
    dnssec-lookaside auto;

    /* Path to ISC DLV key */
    bindkeys-file "/etc/named.iscdlv.key";

    managed-keys-directory "/var/named/dynamic";
};
```

This is a fairly typical DNS configuration file. The `allow-query` setting makes use of the `network_cidr_enp0s8` fact to allow hosts in the same subnet as the server to query the server.

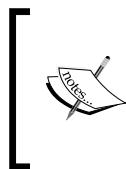
Exported Resources

The named.conf file then includes definitions for the various zones handled by the server, as shown in the following code:

```
zone "." IN {
    type hint;
    file "named.ca";
};

zone "example.com" IN {
    type master;
    file "zone.example.com";
    allow-update { none; };
};
zone "<%= @reverse_enp0s8 -%>" {
    type master;
    file "reverse.<%= @reverse_enp0s8 -%>";
};
```

The zone file headers are defined from templates that use the local time to update the zone serial number.



DNS zone files must contain a **Start of Authority (SOA)** record that contains a timestamp used by downstream DNS servers to determine if they have the most recent version of the zone file. Our template will use the Ruby function Time.now.gmtime to append a timestamp to our zone file.



The zone for example.com is as follows:

```
$ORIGIN example.com.
$TTL1D
@      IN SOA   root hostmaster (
<%= Time.now.gmtime.strftime("%Y%m%d%H") %> ; serial
8H      ; refresh
4H      ; retry
4W      ; expire
1D )    ; minimum
          NS      ns1
          MX      10 ns1
;
; just in case someone asks for localhost.example.com
localhost   A      127.0.0.1
ns1        A      192.168.122.1
; exported resources below this point
```

The definition of the reverse zone file template contains a similar SOA record and is defined as follows:

```
$ORIGIN 1.168.192.in-addr.arpa.  
$TTL1D  
@ IN SOAdns.example. hostmaster.example. (  
<%= Time.now.gmtime.strftime("%Y%m%d%H") %> ; serial  
    28800      ; refresh (8 hours)  
    14400      ; retry (4 hours)  
    2419200    ; expire (4 weeks)  
    86400      ; minimum (1 day)  
)  
NS ns.example.  
; exported resources below this point
```

With all this in place, we only need to apply the `example::dns::server` class to a machine to turn it into a DNS server for `example.com`. As more and more nodes are given the `example::dns::client` class, the DNS server receives their exported resources and builds up zone files. Eventually, when all the nodes have the `example::dns::client` class applied, the DNS server knows about all the nodes under Puppet control within the enterprise. As shown in the following output, the DNS server is reporting our `stand` node's address:

```
[root@stand ~]# nslookup dns1.example.com 192.168.1.54  
Server:          192.168.1.54  
Address:        192.168.1.54#53  
  
Name:      dns1.example.com  
Address:    192.168.1.54  
  
[root@stand ~]# nslookup stand.example.com 192.168.1.54  
Server:          192.168.1.54  
Address:        192.168.1.54#53  
  
Name:      stand.example.com  
Address:    192.168.1.1
```

Although this is a simplified example, the usefulness of this technique is obvious; it is applicable to many situations.

Summary

In this chapter, we installed and configured PuppetDB. Once installed, we used PuppetDB as our storeconfigs container for exported resources. We then showed how to use exported resources to manage relationships between nodes. Finally, we used many of the concepts from earlier chapters to build up a complex node relationship for the configuration of DNS services.

In the next chapter, we will explore a design paradigm that reduces clutter in node configuration and makes understanding the ways in which your modules interact easier to digest.

9

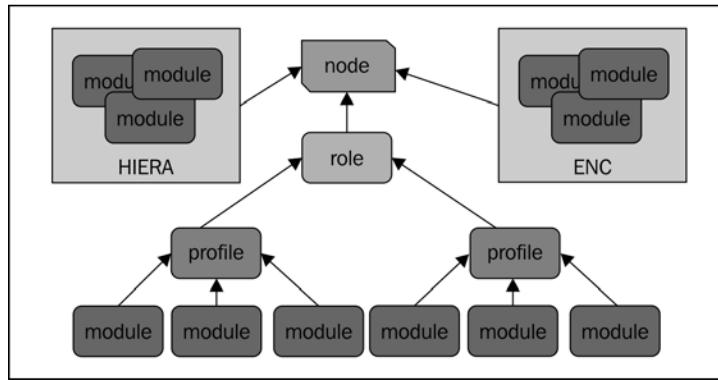
Roles and Profiles

In *Chapter 2, Organizing Your Nodes and Data*, we showed you how to organize your nodes using an ENC or Hiera, or ideally both. At that point, we didn't cover the Forge modules or writing your own modules, as we did in *Chapter 4, Public Modules*, and *Chapter 5, Custom Facts and Modules*. In this chapter, we will cover a popular design concept employed in large installations of Puppet. The idea was originally made popular by Craig Dunn in his blog, which can be found at <http://www.craigdunn.org/2012/05/239/>. Garry Larizza also wrote a useful post on the subject at <http://garylarizza.com/blog/2014/02/17/puppet-workflow-part-2/>.

Design pattern

The concept put forth by Craig Dunn in his blog is the one at which most Puppet masters arrive independently. Modules should be nested in such a way that common components can be shared among nodes. The naming convention that is generally accepted is that roles contain one or more profiles. Profiles in turn contain one or more modules. You can have a node-level logic that is very clean and elegant using the roles and profile design patterns, together with an ENC and Hiera,. The ENC and/or Hiera can also be used to enforce standards on your nodes without interfering with the roles and profiles. As we discussed in *Chapter 2, Organizing Your Nodes and Data*, with the virtual module it is possible to have Hiera apply classes automatically to any system where the `is_virtual` fact is true. Applying the same logic to facts such as `osfamily`, we can ensure that all the nodes for which `osfamily` is RedHat, receive an appropriate module.

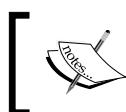
Putting all these elements together, we arrive at the following diagram showing how modules are applied to a node:



Roles are the high-level abstraction of what a node will do.

Creating an example CDN role

We will start by constructing a module for a web server (this example is a cliché). What is a web server? Is a web server an Apache server or a Tomcat server or both, or maybe even Nginx? What file systems are required? What firewall rules should be applied, always? The design problem is figuring out what the commonalities are going to be and where to divide them. In most enterprises, creating a blanket "web server" module won't solve any problems and will potentially generate huge case statements. If your modules follow the roles-and-profiles design pattern, you shouldn't need huge case statements keyed off `$: : hostname`; nodes shouldn't be mentioned in your role module. To elaborate this point further, let's take a look at an example of our companies' **Content Delivery Network (CDN)** implementation. The nodes in the CDN will be running Nginx.



The use of Nginx for CDN is only given as an example. This in no way constitutes an endorsement of Nginx for this purpose.

We'll create an Nginx module, but we'll keep it simple so that it just performs the following functions:

1. Install Nginx.
2. Configure the service to start.
3. Start the service.

To configure Nginx, we need to create the global configuration file, `/etc/nginx/nginx.conf`. We also need to create site configuration files for any site that we wish to include in `/etc/nginx/conf.d/<sitename>.conf`. Changes to either of these files need to trigger the Nginx service to refresh. This is a great use case for a parameterized class. We'll make the `nginx.conf` file into a template and allow some settings to be overridden, as shown in the following code:

```
class nginx (
  Integer $worker_connections = 1024,
  Integer $worker_processes = 1,
  Integer $keepalive_timeout = 60,
  Enum['installed','absent'] $nginx_version = 'installed',
) {
  file {'nginx.conf':
    path      => '/etc/nginx/nginx.conf',
    content   => template('nginx/nginx.conf.erb'),
    mode      => '0644',
    owner     => '0',
    group     => '0',
    notify    => Service['nginx'],
    require   => Package['nginx'],
  }
  package {'nginx':
    ensure => $nginx_version,
  }
  service {'nginx':
    require => Package['nginx'],
    ensure  => true,
    enable   => true,
  }
}
```



The class shown here uses the newer Puppet type syntax and will result in syntax errors on Puppet versions lower than 4.

The `nginx.conf.erb` template will be very simple, as shown in the following code:

```
# HEADER: created by puppet
# HEADER: do not edit, contact puppetdevs@example.com for changes
user  nginx;
worker_processes<%= @worker_processes -%>;
```

```
error_log  /var/log/nginx/error.log;
pid        /var/run/nginx.pid;

events {
    worker_connections<%= @worker_connections -%>;
}
http {
    include  /etc/nginx/mime.types;
    default_type application/octet-stream;
    log_format  main  '$remote_addr - $remote_user [$time_local]
"$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';
    access_log  /var/log/nginx/access.log  main;
    sendfile      on;
    keepalive_timeout<%= @keepalive_timeout -%>;
    include /etc/nginx/conf.d/*.conf;
}
```

Now, we need to create the `define` function for an Nginx server (not specific to the CDN implementation), as shown in the following code:

```
define nginx::server (
    $server_name,
    $error_log,
    $access_log,
    $root,
    $listen = 80,
) {
    include nginx
    file {"nginx::server::$server_name":
        path      => "/etc/nginx/conf.d/${server_name}.conf",
        content   => template('nginx/server.conf.erb'),
        mode      => '0644',
        owner     => '0',
        group     => '0',
        notify    => Service['nginx'],
        require   => Package['nginx']
    }
}
```

To ensure that the autoloader finds this file, we put the definition in a file called `server.pp`, within the manifests directory of the Nginx module (`nginx/manifests/server.pp`). With the defined type for `nginx::server` in hand, we will create a CDN profile to automatically configure a node with Nginx and create some static content, as follows:

```
class profile::cdn{
(
  Integer $listen = 80,
) {

  nginx::server {"profile::nginx::cdn::$::fqdn":
    server_name => "${::hostname}.cdn.example.com",
    error_log   => "/var/log/nginx/cdn-${::hostname}-error.log",
    access_log  => "/var/log/nginx/cdn-${::hostname}-access.log",
    root        => "/srv/www",
    listen      => $listen,
  }
  file {'/srv/www':
    ensure  => 'directory',
    owner   => 'nginx',
    group   => 'nginx',
    require  => Package['nginx'],
  }
  file {'/srv/www/index.html':
    mode    => '0644',
    owner   => 'nginx',
    group   => 'nginx',
    content => @("INDEXHTML"/L)
      <html>
        <head><title>${::hostname} cdn node</title></head>
        <body>
          <h1>${::hostname} cdn node</h1>
          <h2>Sample Content</h2>
        </body>
      </html>
      | INDEXHTML
    ,
    require  => [Package['nginx'],File['/srv/www']],
  }
}
```



The preceding code uses the newer Heredocs syntax of Puppet 4. This is a more compact way to represent multiline strings in Puppet code. More information on Heredocs is available at http://docs.puppetlabs.com/puppet/latest/reference/lang_data_string.html#heredocs.

Now all that is left is to define the role is to include this profile definition, as follows:

```
class role::cdn {
    include profile::cdn
}
```

Now, the node definition for a CDN node will only contain the `role::cdn` class, shown as follows:

```
nodefirstcdn {
    include role::cdn
}
```

Creating a sub-CDN role

Now that we have a `role::cdn` class to configure a CDN node, we will configure some nodes to run Varnish in front of Nginx.



Varnish is a web accelerator (caching HTTP reverse proxy). More information on Varnish is available at <http://www.varnish-cache.org>. In our implementation, Varnish will be provided by the EPEL repository.

In this configuration, we will need to change Nginx to only listen on `127.0.0.1` port 80 so that Varnish can attach to port 80 on the default IP address. Varnish will accept incoming connections and retrieve content from Nginx. It will also cache any data it retrieves and only retrieve data from Nginx when it needs to update its cache. We will start by defining a module for Varnish that installs the package, updates the configuration, and starts the service, as shown in the following code:

```
class varnish
(
    String $varnish_listen_address = "$::ipaddress_eth0",
    Integer $varnish_listen_port      = 80,
    String $backend_host            = '127.0.0.1',
    Integer $backend_port           = 80,
)
```

```

package {'varnish':
    ensure => 'installed'
}
service {'varnish':
    ensure  => 'running',
    enable   => true,
    require  => Package['varnish'],
}
file {'/etc/sysconfig/varnish':
    mode      => '0644',
    owner     => 0,
    group    => 0,
    content   => template('varnish/sysconfig-varnish.erb'),
    notify    => Service['varnish']
}

file {'/etc/varnish/default.vcl':
    mode      => '0644',
    owner     => 0,
    group    => 0,
    content   => template('varnish/default.vcl.erb'),
    notify    => Service['varnish'],
}
}
}

```

Now, we need to create a profile for Varnish, as shown in the following code. In this example, it will only contain the varnish class, but adding this level allows us to add extra modules to the profile later:

```

# profile::varnish
# default is to listen on 80 and use 127.0.0.1:80 as backend
class profile::varnish{
    include ::varnish
}

```



We need to specify `::varnish` to include the module called `varnish`. Puppet will look for Varnish at the current scope (`profile`) and find `profile::varnish`.

Next, we will create the role `cdn::varnish`, which will use `role::cdn` as a base class, as shown in the following code:

```

class role::cdn::varnish inherits role::cdn {
    include profile::varnish
}

```

One last thing we need to do is to tell Nginx to only listen on the loopback device (127.0.0.1). We can do that with Hiera; we'll assign a top scope variable called `role` to our node. You can do this through your ENC or in `site.pp`, as follows:

```
$role = hiera('role', 'none')
node default {
    hiera_include('classes', base)
}
```

Now, create a YAML file for our `cdn::varnish` role at `hieradata/roles/role::cdn::varnish.yaml` with the following content:

```
---
profile::cdn::listen: '127.0.0.1:80'
```

We declared a parameter named `listen` in `profile::cdn` so that we could override the value. Now if we apply the `role::cdn::varnish` role to a node, the node will be configured with Nginx to listen only to the loopback device; Varnish will listen on the public IP address (`::ipaddress_eth0`) on port 80. Moreover, it will cache the content that it retrieves from Nginx.

We didn't need to modify `role::cdn`, and we made `role::cdn::varnish` inherit `role::cdn`. This model allows you to create multiple sub-roles to fit all the use cases. Using Hiera to override certain values for different roles removes any ugly conditional logic from these definitions.

Dealing with exceptions

In a pristine environment, all your nodes with a certain role would be identical in every way and there would be no exceptions. Unfortunately, dealing with exceptions is a large part of the day-to-day business of running Puppet. It is possible to remove node level data from your code using roles and profiles together with Hiera, (roles, profiles, and modules).

Hiera can be used to achieve this separation of code from data. In *Chapter 2, Organizing Your Nodes and Data*, we configured `hiera.yaml` with `roles/%{::role}` in the hierarchy. The defaults for any role will be put in `hieradata/roles/[rolename].yaml`. The hierarchy determines the order in which files are searched for Hiera data. Our configuration is as follows:

```
---
:hierarchy:
  - "zones/%{::example_zone}"
  - "hosts/%{::hostname}"
  - "roles/%{::role}"
```

- "%{::kernel}/%{::osfamily}/%{::lsbmajdistrelease}"
- "is_virtual/%{::is_virtual}"
- common

Any single host that requires an exception to the default value from the `roles` level YAML file can be put in either the `hosts` level or `zones` level YAML files.

The idea here is to keep the top-level role definition as clean as possible; it should only include profiles. Any ancillary modules (such as the `virtual` module) that need to be applied to specific nodes will be handled by either Hiera (via `hiera_include`) or the ENC.

Summary

In this chapter, we explored a design concept that aims to reduce complexity at the topmost level, making your node definitions cleaner. Breaking up module application into multiple layers forces your Puppeteers to compartmentalize their definitions. If all the contributors to your code base consider this, collisions will be kept to a minimum and exceptions can be handled with host-level Hiera definitions.

In the next chapter, we will look at how to diagnose inevitable problems with catalog compilation and execution.

10

Troubleshooting

Inevitably, you will run into problems with your Puppet runs but having a good reporting mechanism is the key to knowing when failures occur. The IRC report mechanism we discussed in *Chapter 7, Reporting and Orchestration*, is useful to detect errors quickly, when most of your Puppet runs are error-free.

If you have more than the occasional error, then the IRC report will just become a noise that you'll learn to ignore. If you are having multiple failures in your code, you should start looking at the acceptance testing procedures. Puppet Labs provides a testing framework known as **Puppet beaker**. More information on Puppet beaker is available at <https://github.com/puppetlabs/beaker>. A simpler option is **rspec-puppet**. More information on rspec-puppet is available at <http://rspec-puppet.com/tutorial/>.

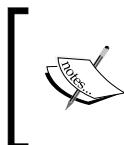
Most of the Puppet failures I've come across end up in two buckets. These buckets are, as follows:

- Connectivity to Puppet and certificates
- Catalog failure

We'll examine these separately and provide some methods to diagnose issues. We will also be covering debugging in detail.

Connectivity issues

As we have seen in *Chapter 1, Dealing with Load/Scale*, at its core, Puppet communication is done using a web service. Hence, whenever troubleshooting problems with Puppet infrastructure, we should always start with that mindset. Assuming you are having trouble accessing the Puppet master, Puppet should be listening on port 8140, by default.



This port is configurable; you should verify the port is 8140 by running the following command:

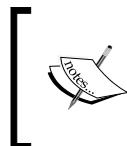
```
# puppet config print masterport  
8140
```



Previous versions of Puppet were run as Passenger processes, under Apache. If you cannot reach your `puppetserver` on port 8140, you may need to check that Apache is at least running.

You should be able to successfully connect to `masterport` and check that you get a successful connection using **Netcat** (`nc`):

```
[root@client ~]# nc -v puppet.example.com 8140  
Ncat: Version 6.40 ( http://nmap.org/ncat )  
Ncat: Connection refused.  
[root@client ~]# nc -v puppet.example.com 8140  
Ncat: Version 6.40 ( http://nmap.org/ncat )  
Ncat: Connected to 192.168.1.1:8140.  
Ncat: 0 bytes sent, 0 bytes received in 2.93 seconds.  
[root@client ~]#
```

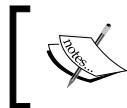


Netcat can be used to check the connectivity of TCP and UDP sockets. If you do not have Netcat (`nc`) available, you can use Telnet for the same purpose. To exit Telnet, issue `Control-]` followed by `quit`.



To exit Netcat after the successful connection, type `Control+D`. If you don't see **succeeded!** in the output, then you are having trouble reaching the `puppetserver` on port 8140. For this type of error, you'll need to check your network settings and diagnose the connection issue. The common tools for that are **ping**, which uses ICMP ECHO messages, and **mtr**, which mimics the traceroute functionality. Don't forget your host-based firewall (`iptables`) rules; you'll need to allow the inbound connection on port 8140.

Assuming that the previous connection was successful, the next thing you can do is use `wget` or `curl` to try to retrieve the CA certificate from the Puppet master.



`wget` and `curl` are simple tools that are used to download information using the HTTP protocol. Any tool that can communicate using HTTP with SSL encryption can be used for our purpose.

Retrieving the CA certificate and requesting a certificate to be signed are two operations that can occur without having certificates. Your nodes need to be able to verify the Puppet master and request the certificates before they have had their certificates issued. We will use `wget` to download the CA certificate, as shown in the following screenshot:

```
[root@client ~]# curl -k https://puppet.example.com:8140/puppet-ca/v1/certificate/ca
-----BEGIN CERTIFICATE-----
MIIFfjCCA2agAwIBAgIBATANBgkqhkiG9w0BAQsFADAoMSYwJAYDVQQDDB1QdXBw
ZXQgQ0E6IH81cHB1dC51eGFtcGx1LmNvbTAeFw0xNTA5MTAwNTI2MTVaFw0yMDA5
MDkwNTI2MTVaMCgxJjAkBgNVBAMMHVB1cHB1dCBDQTogcHVwcGV0LmV4YW1wbGUu
Y29tMIICIJANBgkqhkiG9w0BAQEFAAOCAg8AMIICgKCAGEAh4TW/ge8Ppj//EA1
c8UePhn2XyG0w3UAeF1o7ptGBEdmY1inG+P7QgbDk44ySzVqyI3WpD1I22qZGXnS
DYhNq7NsKnNgJftGE4MphYiqBzu0tk0g0SqnGFe01YYBNPFEiPcA6RGPi3YNfe1P
Fd8w1kMubjLfIoFtDG6AZHW9m08j9gA1U0PDrfgKy9Ye71oL6DYRMaUp9MpFAq0
tQr+uoAH/LS5EXam7+DRk0c1MCRddE80UmtR8RjsFSKMuQL2C0gw5hLNCLDEcox
```

Another option is using `gnutls-cli` or the OpenSSL `s_client` client programs. Each of these tools will help you diagnose certificate issues, for example, if you want to verify that the Puppet master is sending the certificate you think it should.

To use `gnutls-cli`, you need to install the `gnutls-utils` package. To connect to your Puppet master on port 8140, use the following command:

```
# gnutls-cli -p 8140 puppet.example.com --no-ca-verification
Resolving 'puppet.example.com'...
Connecting to '192.168.1.1:8140'...
- Successfully sent 0 certificate(s) to server.
...
- Simple Client Mode:
```

You will then have an SSL-encrypted connection to the server, and you can issue standard HTTP commands, such as GET. Attempt to download the CA certificate by typing the following command:

```
GET /puppet-ca/,v1/certificate/ca HTTP/1.0
Accept: text/plain
```

Troubleshooting

The CA certificate will be returned as text, so we need to specify that we will accept a response that is not HTML. We will use `Accept: text/plain` to do this. The CA certificate should be exported following the HTTP response header, as shown in the following screenshot:

```
- Version: TLS1.2
- Key Exchange: RSA
- Cipher: AES-128-CBC
- MAC: SHA1
- Compression: NULL
- Handshake was completed

- Simple Client Mode:

GET /puppet-ca/v1/certificate/ca HTTP/1.0
Accept: text/plain

HTTP/1.1 200 OK
Date: Wed, 16 Dec 2015 06:21:46 GMT
X-Puppet-Version: 4.2.1
Content-Type: text/plain; charset=ISO-8859-1
Content-Length: 1964
Server: Jetty(9.2.z-SNAPSHOT)

-----BEGIN CERTIFICATE-----
MIIFfjCCA2agAwIBAgIBATANBgkqhkiG9w0BAQsFADoMSYwJAYDVQQDDB1QdXBw
ZXQgQ0E6IHB1cHB1dC5leGFtcGx1LmNvbTAeFw0xNTA5MTAwNTI2MTVaFw0yMDA5
MDkwNTI2MTVaMCgxJjAkBgnVBAMMHVB1cHB1dCBDQTogcHwvcGV0LmV4Yw1wbGUu
Y29tMIICijANBgkqhkiG9w0BAQEFAAOCAg8AMIIICgKCAgEAh4Tw/ge8Pqj//EA1
c8UePnh2XyG0w3UaEFl07ptGBEdmYJinG+P7QgbDk44ySzVqyI3WpD1I22qZGXnS
DYhNq7NsKnNgJftGE4MpHYiqBzuDtk0g0SqnGFe01YYBNPFEiPcA6RGPi3YNfe1P
Fd8wlkMubjLfIoFtDG6AZHVW9m08j9gA1UOPDrfgKy9Ye71oL6DYZRMaUp9MpFAq0
tQr+uoAH/LS5EXam7+DRk0c1MCRddE80UmtR8RjsFSKMguQL2C0gw5hLNCLDEcox
```

Using OpenSSL's `s_client` program is similar to using `gnutls-cli`. You will need to specify the host and port using the `-host` and `-port` parameters or `(-connect hostname:port)`, as follows (`s_client` has a less verbose mode, `-quiet`, which we'll use to make our screenshot smaller):

```
[root@client ~]# openssl s_client -connect puppet.example.com:8140 -quiet
depth=0 CN = puppet.example.com
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 CN = puppet.example.com
verify error:num=27:certificate not trusted
verify return:1
depth=0 CN = puppet.example.com
verify error:num=21:unable to verify the first certificate
verify return:1
GET /production/certificate/ca HTTP/1.0
Accept: text/plain

HTTP/1.1 200 OK
Date: Wed, 16 Dec 2015 06:29:01 GMT
X-Puppet-Version: 4.2.1
Content-Type: text/plain; charset=ISO-8859-1
Content-Length: 1964
Server: Jetty(9.2.z-SNAPSHOT)

-----BEGIN CERTIFICATE-----
MIIFfjCCA2agAwIBAgIBATANBgkqhkiG9wOBAQsFADoMSYwJAYDVQQDDB1QdXBw
ZXQgQ0E6IHB1cHBldC5leGFtcGxLmNvbTAeFw0xNTA5MTAwNTI2MTVaFw0yMDA5
```

Catalog failures

When the client requests a catalog, it is compiled on the master and sent down to the client. If the catalog fails to compile, the error is printed and can, most likely, be corrected easily. For example, the following base class has an obvious error:

```
class base {
  file {'one':
    path    => '/tmp/one',
    ensure  => 'directory',
  }
  file {'one':
    path    => '/tmp/one',
    ensure  => 'file',
  }
}
```

Troubleshooting

The file resource is defined twice with the same name. The error appears when we run Puppet, as shown in the following screenshot:

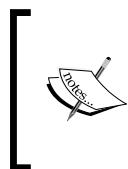
```
[root@client ~]# puppet apply base.pp
Error: Evaluation Error: Error while evaluating a Resource Statement, Cannot alias File[two] to ["/tmp/one"] at /root/base.pp:6; resource ["File", "/tmp/one"] already declared at /root/base.pp:2 at /root/base.pp:6:3 on node client.example.com
[root@client ~]#
```

Fixing this type of duplicate declaration is very straightforward; the line numbers of each declaration are printed in the error message. Simply locate the two files and remove one of the entries.

A more perplexing issue is when the catalog compiles cleanly but fails to apply on the node. The catalog is stored in the agent's `client_data` directory (current versions use JSON files, earlier versions used YAML files). In this case, the file is stored in `/opt/puppetlabs/puppet/cache/client_data/catalog/client.example.com.json`. Using `jq`, we can examine the JSON file and find the problem definitions.

`jq` is a JSON processor and is available in the EPEL repository on enterprise Linux installations.

```
[root@client catalog]# jq '.resources[] .title' <client.example.com.json
"main"
"Settings"
"Main"
"default"
"Base"
"one"
```



You can always just read the JSON file directly, but using `jq` on extremely large files is useful. You can use `jq` as you would use `grep` on a file, thus making searching within a JSON file much easier. More information on `jq` can be found at <http://stedolan.github.io/jq/>.

Now, to look at our problem definition, we'll select the resource whose title is "one", as shown here:

```
[root@client catalog]# jq '.resources[] | select(.title=="one")' <client.example.com.json
{
  "type": "File",
```

```
"title": "one",
"tags": [
  "file",
  "one",
  "class",
  "base",
  "node",
  "default"
],
"file": "/etc/puppetlabs/code/environments/production/modules/base/
manifests/init.pp",
"line": 17,
"exported": false,
"parameters": {
  "path": "/tmp/one",
  "ensure": "directory"
}
}
```

You may force a master to compile a catalog for a node, as follows (Puppet will print out the catalog, in JSON format, to the terminal):

```
[root@stand ~]# puppet master --compile client.example.com
Notice: Compiled catalog for client.example.com in environment production
in 0.60 seconds
{
  "tags": ["settings", "default", "base", "node", "class"],
  "name": "client.example.com",
  "version": 1450506795,
  "environment": "production",
  "resources": [
    ...
  ]
}
```

Full trace on a catalog compilation

Using `puppet master --compile`, you can also select to run a full trace on the compilation with the `--trace` option. This option will show which providers were run and a much higher level of detail than the debug output. To do so, specify the log destination as well. Running a full trace will generate a lot of data and you'll want to store that in a log file.

```
[root@stand ~]# puppet master --compile client.example.com
Notice: Compiled catalog for client.example.com in environment production in 0.60 seconds
{
  "tags": ["settings", "default", "base", "node", "class"],
  "name": "client.example.com",
  "version": 1450506795,
  "environment": "production",
  "resources": [
    {
      "type": "Stage",
      "title": "main",
      "tags": ["stage"],
      "exported": false,
      "parameters": {
        "name": "main"
      }
    },
    {
      "type": "Class",
      "title": "Settings",
    }
  ]
}
```

The following output shows that we can see a lot more information than what the normal `--debug` flag will show. The log file will also compile the catalog in the production environment by default:

```
[root@stand ~]# head /var/log/puppetlabs/client.example.com.log
2015-12-19 01:36:29 -0500 Puppet (debug): Applying settings catalog for sections main
, master, ssl, metrics
2015-12-19 01:36:30 -0500 Puppet (debug): Evicting cache entry for environment 'produ
ction'
2015-12-19 01:36:30 -0500 Puppet (debug): Caching environment 'production' (ttl = 0 s
ec)
2015-12-19 01:36:30 -0500 Puppet (debug): Evicting cache entry for environment 'produ
ction'
2015-12-19 01:36:30 -0500 Puppet (debug): Caching environment 'production' (ttl = 0 s
ec)
2015-12-19 01:36:30 -0500 Puppet (debug): Evicting cache entry for environment 'produ
ction'
```

To compile for another environment, specify the environment with `--environment`, as shown in the following command:

```
[root@stand ~]# puppet master --compile client.example.com --debug  
--trace  
--logdest /var/log/puppetlabs/client.example.com.log --environment  
sandbox
```

The classes.txt file

The `/opt/puppetlabs/puppet/cache/state/classes.txt` file contains a list of classes applied to the machine. If you are having trouble with a node, you can search here for the last set of classes that were successfully applied to a node. But, when you are having trouble, you are most interested in the classes in the current catalog and the classes that are different or missing. We can use `jq` again to query the JSON of the current catalog, as shown in the following command:

```
[root@client ~]# jq .classes[] </opt/puppetlabs/puppet/cache/client_data/  
catalog/client.example.com.json  
"settings"  
"default"  
"base"
```



Settings and default are classes that are internal to Puppet and not user-defined. In this output, only the base was defined by our manifests.



We can compare the list of classes returned by `jq` to those listed in `classes.txt`. The classes shown in `classes.txt` are from the last successful run of Puppet. The file is created at the end of the Puppet agent run. The classes returned by `jq` are from the catalog, which just fails to apply if we are debugging. These two lists will be consistent on a node with a successful Puppet agent run.

Debugging

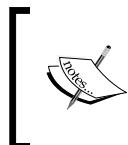
Turning on the debugging option on your Puppet master isn't such a big deal with a few hundred nodes. However, in an environment with thousands of nodes, it isn't a viable option. Nevertheless, you sometimes need to enable debugging to figure out where catalog compilation is failing. Our proxy configuration comes to the rescue here. The idea is to have one Puppet master dedicated to debugging. The debugging server will have debugging turned on, by changing the `puppetserver` logging settings in the `logback.xml` file. The advantage of this method over that of running `puppet master -compile`, as we showed earlier, is that, while you are debugging your node, you place it in a debugging environment (`problem` for instance). While the node is in the debugging environment, it will be removed from your reporting infrastructure and not continue to alert you to failures.

To do this, we go back to our `proxy.conf` file on our Puppet master and define a new balancer named `puppetproblem` that goes to our debugging worker. We'll use `worker2` (192.168.100.102) in the following example:

```
<Proxy balancer://puppetproblem>
    BalancerMember http://192.168.100.102:18140
</Proxy>
```

We now add a new `ProxyPassMatch` line to our `VirtualHost` right after the certificate matching line:

```
ProxyPassMatch ^/(problem/.*)$ balancer://puppetproblem/$1
```



Whenever we add a new `ProxyPassMatch` line to the `proxy.conf` file, make sure that the first entry is always the certificate matching line. If you place anything before the certificate line, certificate requests will not be routed to your CA machines.

Restart `httpd` on the master to make the change effective. With this in place, we edit `logback.xml` on our debugging Puppet master and change the `LOGLEVEL` to `DEBUG`.

Restart `puppetserver` on the debugging Puppet master to make the change effective. Now, when you have a problem with a node, you can send it to `worker2` by specifying the environment "problem" when running the agent. The steps to diagnose a problem are, as follows:

1. Create the problem branch in Git.
2. Work on the issue.
3. Set the environment of a test node to the new environment.

4. Solve the problem.
5. Merge that branch back into the working branch or production.

Using this method, you can also tie the catalog compilation to a specific worker, which makes tracking down bugs much easier. Without this, your catalog might compile on any one of your workers and some large installations have several workers.

Personal and bugfix branches

When working through a catalog compilation issue, it is sometimes useful to start attacking the problem and changing things on-the-fly. To avoid problems with other nodes, you should work in a new branch (which will create a new environment, just as we configured our Puppet masters to have dynamic environments in *Chapter 3, Git and Environments*). If you are frequently creating branches, you can create one named after yourself or your username, for instance. In an example in *Chapter 3, Git and Environments*, we created a `thomas` branch and worked in the `thomas` branch by specifying `--environment thomas` when running `puppet agent`. Working through problems in a personal branch is a great troubleshooting technique that allows the rest of the nodes to continue working against the main branch or master. If multiple members of your team are working on an issue, it is useful to create a working branch for your team, possibly named either after the issue or more likely after the trouble ticket created by the issue.

Echo statements

When working on a problem branch, you are free to add any number of debugging print or echo statements to your code. In Puppet, these take the form of `notice` or `notify` lines. I prefer `notify` lines, since `notify` lines will be printed when I run `puppet agent -t` on a node. Usually, I place all the variables of the affected module in a single `notify` statement to make sure that the variables are getting set to the values I believe they should. This method is very useful when working with data from Hiera, where you would like to know if the value returned by Hiera is correct, as shown in the following example:

```
$importantSetting = hiera('importantSetting', 'defaultValue')
notify {"importantSetting is $importantSetting": }
```

It is not uncommon to have many `notify` lines throughout a module during the development phase.

Scope

Occasionally, you will have naming conflicts with variables or modules when working on a large code base. For variables, using a notify statement can quickly determine if your code is using the variable you believe it should. For modules, it can sometimes be difficult to determine if the module you intended is being included. For example, you have two modules called packages and example::ntp::packages. The packages module contains a single notify statement in packages/manifests/init.pp, as shown in the following code:

```
class packages {  
    notify {"this is packages":}  
}  
}
```

The example::ntp::packages module has a similar notify statement in example/manifests/ntp/packages.pp, as shown in the following code:

```
class example::ntp::packages {  
    notify {"this is example::ntp::packages": }  
}
```

Now, in example/manifest/ntp.pp, we use include packages, as shown in the following code:

```
class example::ntp {  
    include packages  
}
```

You may be surprised by the following result from puppet agent:

```
# puppet agent -t  
...  
Notice: this is example::ntp::packages  
  
Notice: /Stage[main]/Example::Ntp::Packages/Notify[this is  
example::ntp::packages]/message: defined 'message' as 'this is  
example::ntp::packages'
```

We might have expected `include` packages to use the top-scope packages class, but it actually searched the local scope and used `example::ntp::packages` instead. When working in a large environment, it is advisable to use very specific names for classes or always specify the scope. We can achieve the result we expected using the following code for the definition of `example::ntp`:

```
class example::ntp {
    include ::packages
}
```

If we run `puppet agent` against this version, we see the notification we were expecting, as follows:

```
# puppet agent -t
...
Notice: this is packages

Notice: /Stage[main]/Packages/Notify[this is packages]/message: defined
'message' as 'this is packages'
```

Profiling and summarizing

If your Puppet runs are taking a long time to complete, it is useful to see where there are bottlenecks. From the command line, you can pass the `--evaltrace --summarize` option to `puppet agent` to tell the agent to keep a track of how long the operations took to complete and display a summary at the end of compilation, as shown in the following screenshot:

```
Info: /Filebucket[puppet]: Starting to evaluate the resource
Info: /Filebucket[puppet]: Evaluated in 0.00 seconds
Notice: Applied catalog in 0.09 seconds
Changes:
Events:
Resources:
    Total: 8
Time:
    Filebucket: 0.00
        File: 0.00
        Schedule: 0.00
    Config retrieval: 1.08
        Total: 1.08
        Last run: 1450508888
Version:
    Config: 1450508887
    Puppet: 4.2.1
[root@client puppetserver]#
```

puppetserver also has the ability to send profiling information to a graphite server. Information on configuring puppetserver to communicate with a graphite server is available at http://docs.puppetlabs.com/pe/latest/puppet_server_metrics.html.

Summary

In this chapter, we examined a few troubleshooting techniques that are useful in the enterprise. Troubleshooting basic network and system connectivity is the first thing to be checked. Using Puppet's Rest API, we were able to talk directly to the master with the help of HTTP tools, such as wget and gnutls-cli. We learned how to read the catalog and use jq to search the catalog on the client. Finally, we showed a method of enabling the expensive debugging feature for specific nodes by creating a debugging worker and directing nodes to that specific worker.

In this book, we took advantage of Puppet's Rest API to scale out our Puppet infrastructure in order to accommodate a large number of nodes. Working in the enterprise, the division of code from data is important to allow modules to be reused and to reduce complexity. A large number of nodes will introduce its own set of complexities. Working to reduce the complexity in your environment will allow you to grow and adapt quickly. Keeping your code as simple as possible will make it easier to find problems when they appear. A large number of nodes creates a level of complexity on its own. As you grow your environment, you should continually look for ways to reduce the quantity and complexity of your code.

Bibliography

This course is a blend of different projects and texts all packaged up keeping your journey in mind. It includes the content from the following Packt products:

- *Puppet 4 Essentials - Second Edition* by Felix Frank and Martin Alfke
- *Extending Puppet - Second Edition* by Alessandro Franceschi and Jaime Soriano Pastor
- *Mastering Puppet - Second Edition* by Thomas Uphill

Index

A

Abstract Syntax Tree (AST) 448
ACLs
 reference link 521
ActiveMQ
 client, connecting to 701, 702
 installing 695, 696, 697
agent
 about 225
 enhancing, through plugins 118, 119
 resources, exporting to 144
All-in-One (AIO) packaging 160
allowed characters 248
Amazon Machine Image (AMI) 488
Amazon Web Services
 about 487
 cloud provisioning 487, 488
anchor pattern 91 341
Ansible
 about 705
 references 587
antipatterns
 avoiding 155, 156
Apache proxy 514, 515, 516
apply command 422
Arista Networks
 reference 484
arrays
 handling 204, 206
array values
 exploiting, defined types used 81, 83
augeas
 about 557
 URL 347

automatic parameter lookup 271
automation tasks, existing infrastructure
 application deployments, refining 384
 common systems
 configurations, automating 384
 integration 385
 monitoring infrastructure, automating 385
 roles, automating 384
 servers deployment, automating 383
AWS Identity and Access Management (IAM) interface
 reference link 488
AWS module
 reference link 489
AWS provisioning
 about 489
 CloudFormation, managing 490

B

backend Puppet Masters 434
backend settings, hiera.yaml configuration file
 *datadir 262
 *datasource 262
beaker
 references 107
beaker-rspec
 URL 412
 using 412, 413, 414
best practices, modules 104
BitBucket
 URL 407
branching models
 URL 581

branching workflow
 URL 581
bugfix branch 753
built-in variables 243

C

CDN role
 example, creating 734-738
cached mode 495
caching 438
ca.conf file 227
catalog 225, 448
catalog compilation
 about 225, 508, 509
 full trace 750
catalog failures 747-749
catalog storage 707
central firewall
 maintaining 150
certificate authority (CA)
 about 516
 in Clojure 500
Certificate Revocation List (CRL) 432, 685
certificates
 managing 431, 432, 433
certificate signing 508
Certificate Signing Requests (CSR) 432
CFacter 656
circular dependencies
 avoiding 18, 19, 20
classes
 about 68, 224, 329
 declaring 68, 69, 233, 234, 235
 defining 68, 69, 233, 319
 defining, for each node 314, 315
 events, passing between 85, 86
 including, from defined types 84
 making, flexible through parameters 92, 93
 versus defined types 72
classes.txt file 751
class inheritance 236
Classless Inter-Domain Routing (CIDR) 724
class names 340
class parameters 336, 337

class parameter values
 binding, automatically 202, 203, 204
client
 connecting, to ActiveMQ 701, 702
Clojure
 about 222, 429
 certificate authority (CA) 500
CloudFormation
 about 490
 reference 490
cloud provisioning
 on Google Compute Engine 490
clustered filesystem
 using 522
CN (common name) 161
code
 data separation 323, 324
code and data
 managing 435
code management
 with Git 402, 403
code optimization 443, 444
code review
 about 406
 Gerrit 406
 online resources, for peer review 407
collector
 about 645
 used, for realizing resources 143, 144
command line
 used, for querying puppetdbquery module 307
Command/Query Responsibility Separation (CQRS) 296
commands
 about 285
 deactivate node 300
 replace catalog 299
 replace facts 299
 store report 300
 using 299, 300
comments
 in modules 641, 642, 643
comparison operators 249
component (application) modules 359, 360

component classes
writing 75, 76
components 311, 312
components, 318
composite design 74
comprehensive classes
about 77
writing 74, 75
concat
URL 346
concat module 612-617
conditionals 248, 249, 337
Config retrieval time 442
configuration data
structuring, in hierarchy 195, 197
configuration directory 350
configuration files
defining, for each node 318
placing, in custom fileserver 319
placing, in Hiera 318
placing, in shared modules 318
placing, in site module(s) 318
configuration hash patterns
managing 348, 350
configuration management
importance 223, 224
Configuration Management Database (CMDB) 723
configuration, PuppetDB
configuration settings 288-293
init script configuration 288
logging configuration 293
on Puppet Master 293, 294
connectivity issues 744-746
contained class 92
container builds 494
container relationships
performance implications 90
containers
limitations 88, 89
ordering 87
relationships, establishing among 84
contain function 92
Content Delivery Network (CDN) 734
Continuous Integration (CI) tool
about 417
Jenkins 419
Travis 418, 419
control structures
adding, in manifests 9, 10
core resource types
reference link 62
C++ prototype
reference link 502
create_resources approach
advantages 209
cron
configuring, Puppet resource used 530, 531
Puppet agent, running from 41
cron resource type 27
curl 745
custom configuration items
automating 148
custom faces 472-475
custom facts
about 53, 459, 647
creating 647, 648, 650-653
creating, for use in Hiera 654, 655, 656
external facts 462, 463
Facter, extending with 53- 56
Ruby facts 460, 461
custom fileserver
configuration files, placing 319
custom functions
about 98, 456
defining 456-458
module interface, refining through 128, 129
custom providers 463-469
custom report handlers 470, 471
custom resources
defining 319
custom resources, locations
Hiera 319
shared modules 319
site module(s) 319
custom types 463-469

D

dashboards
about 294
performance dashboard 294, 295
Puppetboard 296

data
resources, converting to 207-210

data binding 234, 271, 317, 455

data, defining in manifest
consequences 193, 194, 195

data separation
about 312
from code 323, 324
in modules 339
reference link 339

data types
about 660, 661
reference link 661

Dead Letter Office 289

debugging 752

default Puppet Master 423, 424

defined resource type 235

defined type, replacing with native type
about 120
basic functionality, implementing 124, 125
management commands, declaring 123
provider, adding 123
provider, allowing to prefetch
existing resources 125, 126
resource names, using 122
resource type interface, creating 121, 122
sensible parameter hooks, designing 122
type, naming 120
type robust, making
during provisioning 127

defined type
about 68, 235, 661-672
classes, including from 84
creating 69-71
events, passing between 85, 86
used, for exploiting array values 81, 82
using 69-71
using, as macros 80
using, as resource multiplexers 79
using, as resource wrappers 77, 78
versus classes 72

dependencies
declaring 14-17
managing 245, 246, 352, 353

design pattern 733, 734

directory environments
defining 567-570

Direct Puppet
about 495, 498
file sync 499

Distinguished Name (DN) 434

DNS SRV records 436

Docker
about 493
URL 493

documentation, in modules 100, 101

Domain Specific Language (DSL) 4, 224

dynamic configuration files
templating 136

dynamic facts 461

dynamic scoping 155 245

E

echo statements 753

Eclipse 400

Embedded Ruby (ERB)
about 638
reference link 638

encryption plugins
about 275
hiera-gpg 275-277

Enterprise Linux 7 (EL7) 512

environment directories
using 161

environment locations
configuring 103

environments
about 561-564
maintaining 101, 102
multiple hierarchies 564, 565
setting up, post-receive used 582-585
single hierarchy 566

environments, Git workflows
branch based automatic environments 405
reference link 405
simplified developer workdir environments 405

EPEL 225

events
 passing, between classes 85, 86
 passing, between defined types 85, 86

exceptions
 dealing with 740

exec resource type 25, 26

existing infrastructure
 coding 387-389
 modifications, applying 390, 391
 node migration 376
 node update 376
 planning, for automation 381, 382
 priorities, defining 382-385
 refactoring 380
 solutions, evaluating 386, 387
 step-by-step process, for automation 380, 381
 tasks, for automation 383

exported resource
 about 715
 collecting 716
 declaring 716

exported resources
 about 251, 284
 virtual resources 252

exported SSH keys 719

expressions combinations 251

external components
 ENC 436
 Exported Resources 437
 Report 437

external fact
 about 239, 462, 463
 reference link 127
 using 56, 57

External Node Classifier (ENC)
 about 210, 233, 259, 312, 535, 688
 example, creating 536-539
 extendibility 450
 hostname strategy 539, 540
 LDAP backend 545
 nodes, organizing with 535, 536
 user variables 242

Extra Packages for Enterprise Linux (EPEL)
 URL 688

F

faces
 enhancements 501

facter
 about 222, 224, 454, 656
 extending, with custom facts 53-56
 goals 58
 systems, summarizing with 49, 50, 51

facts
 about 224, 238, 316
 external facts 239
 system knowledge, enhancing through 127
 system's facts 238

facts.d directory
 external facts 462, 463

facts driven approach
 reference link 332

fact values
 accessing 51, 52, 53
 using 51, 52, 53

features 61

files
 about 329
 class names 340
 configuration hash patterns, managing 348, 350
 dependencies, managing 352, 355
 extra resources, managing 354, 355, 356
 installation options, managing 353, 354
 managing 346, 348
 multiple configuration files, managing 350, 351
 restoring, from filebucket 256, 257
 users, managing 352, 353

files-based logic 363

file sync 495, 499

file system
 changes, verifying 439

filesystem access control lists (FACLs) 609

firewalld 623

firewall module 623-627

Foreman
 about 311, 313, 319, 688
 attaching, to Puppet 689, 690, 691

features 320
installing 688
using 691, 692

foreman-report file
download link 690

Forge
modules 602, 603
URL 132, 599

Forge module
finding 132
using 707-710

frontend servers 434

full trace, of catalog compile 750

Fully Qualified Domain
Name (FQDN) 52, 227, 535

functions
learning 172-175

G

Gatling
URL 442

gem 261

generalization
avoiding 105

general purpose define 350, 351

Gepetto
about 222, 400
URL 400

Gerrit
about 406
URL 407

gfs2 522

Git
about 522, 570
defining 571
URL 571
using 595-597

Git documentation
URL 571

GitEnterprise
URL 407

Git hooks
about 403, 581
commits, controlling 591-594
git/hooks/post-receive 404
git/hooks/pre-commit 404

git/hooks/pre-receive 404
post-receive used, for setting up environments 582-585
puppet-sync 585-587
URL 404
used, with developers 588, 589, 590

GitHub
URL 407
using, for public modules 599-601

git ready
URL 403

Git workflow
about 402, 572-581
branches 404
code management, with Git 402, 403
environments 404
Git hooks 403, 404

global.conf file 227

global settings, hiera.yaml configuration file
*backends 262
*hierarchy 262
*logger 262
*merge_behavior 262

glue components 502

glusterfs 522

gpg private key 277

grouping classes 325

H

hammer-cli command
about 321
URL 321

hash
dealing with 270, 271

hashes
handling 204, 206

hash merge
reference link 268

HEREDOC
Multiline, handling with 180-182

Heredocs
reference link 738

Hiera
about 222, 260, 313, 550, 561-564
configuration files, placing 318

configuring 197, 198, 260, 261, 551-554
custom resources 319
extendibility 450
hash, dealing with 270, 271
hiera_include, using 554-560
installing 260, 261
Puppet 3 automatic parameter lookup 271,
 272
shared modules 319
site module(s) 319
URL 550
usage patterns, evolving for class parameters 272, 273
user variables 242
using 269
using, as ENC 280

Hiera backend
about 273, 308, 309
encryption plugins 275
hiera-eyaml 277-279
hiera-file 273, 274, 275
hiera-http 279
hiera-mysql 279

Hiera data
storing 198, 199

hiera-eyaml 277-279

hiera-file
about 273-275
URL 273

Hiera functions, parameters
second argument 270
third argument 270

hiera-gpg
about 275-277
URL 275

hiera-http
about 279
URL 279

Hiera lookups
debugging 214

hiera-mysql
about 279
URL 279

hierarchy
configuration data, structuring in 195, 197

Hiera values
retrieving 200
using, in manifests 200

hiera.yaml configuration file
about 261
backend specific settings 262
command line, working with 264-268
examples 262-264
global settings 262

host entry 716, 718

hostname strategy
used, for modified ENC 541-544

hosts files
managing, locally 147, 148

hosts provisioning 493

HTTP APIs
reference link 508

httpd
about 514

I

Icinga 149

idempotence 225

identifying parameters 316

immutability 11

include keyword
preferring 94

indirector
about 452-456
URL 454

Infrastructure as code 391

Infrastructure as Code paradigm 107

infrastructure automation
approaches, examining 373, 374
existing automated
 infrastructures, updating 378-380
existing infrastructures, automating 375,
 376
modifications 391-394
new infrastructures, creating 374, 375
potential scenarios, examining 373, 374
Puppet friendly infrastructure, managing
 394, 395
Puppet-friendly software, using 395, 396

infrastructure logic 389
infrastructure optimizations
 about 437
 caching 438
 file system changes, verifying 439
 Puppet run times, distributing 438, 439
 stored configs, scaling 439-441
 traffic compression 437
inifile module 618-623
installation, Puppet 4, 5
installing
 ActiveMQ 695-697
 Foreman 688
 PostgreSQL 712
 Puppet 711
 PuppetDB 711
Interactive Ruby (IRB) 648
Internet Relay Chat (IRC) 683-687
inventory service 284
iptables 623
isolation technologies
 Capabilities 492
 Control Groups 492
 Namespaces 492
iteration
 about 250
 expressions combinations 251
 in operator 251
iterator functions
 using 83

J

Jenkins 419
Jenkins plugins
 GitHub / Gerrit 419
 RVM / rbenv 419
 ssh 419
 URL 419
 Vagrant 419
jq
 reference link 748
jruby-ldap module
 reference 549
JSON 495

Juniper
 about 483
 modules 483
Junos OS 483
K
Kermit 223
L
lambdas
 about 250
 learning 172, 173, 174, 175
LDAP
 about 313
 reference 548
 URL 313
LDAP backend
 about 545
 OpenLDAP configuration 545-550
Librarian
 about 604
 using 604, 605, 606
librarian-puppet
 about 401
 Puppet code, deploying with 415
 reference link 604
 URL 415
load balancing
 options 435, 436
load balancing machine
 Puppet certificate authority, splitting off from 523
local repository
 updating 601
logback
 about 683
 reference link 683
logical volume manager (lvm) 627-629
M
macros
 defined types, using as 80
mandatory access controls (MAC) 517

manifest
about 98, 224
control structures, adding in 9, 10
dry testing 9
Hiera values, using in 200
manifest and Hiera designs
selecting 210, 211
master
configuring 518-520
configuring, to store exported resources 146, 147
Master 225
masterless configuration 507, 527
masterless Puppet 436, 437
Master-of-Master (MoM) 536
mcollective
about 222, 681, 693, 694
reference link 694
using 704, 705
reference link 417
Mellanox
reference 484
MessagePack 495
Message Queue (MQ) 693
metaparameters
about 14, 245, 718
reference link 718
model
substantiating, with providers 61, 62
modify on collect 726
mod_passenger 424
mod_rails 424
mod_ssl package 514
module files 637
module functionality
implementing 109-112
module interface
refining, through custom functions 128, 129
module manifest files 634-636
modules
about 224, 253, 335, 634
auto loading 253, 255
best practices 104
building 108
characteristics, identifying 132, 133
creating, with Puppet module 639, 641
data separation 339
documentation 100, 101
implementing 105
installing 103, 104
making, available for Puppet 108
making, portable across platforms 130
naming 108, 638
obtaining 103, 104, 599
overview 98
parts 98
path 253, 255
practical example 211-213
reference link 100
safe testing, with environments 107
structure 99, 100
templates 255, 256
testing 106
Modules 223
modules, Forge 602, 603
monolithic implementation 74
mount resource type 27
Mozilla SSL Configuration Generator
URL 427
mtr 744
Multiline
handling, with HEREDOC 180-182
multi-master scaling
about 430
certificates, managing 431-433
code and data, managing 435
SSL termination, managing 433-435
multiple configuration files
managing 350, 351
multiple definitions 644-646
Murder
URL 437

N

Nagios 149
Nagios configuration
simplifying 149
name variable 122
native Puppet, on network equipment
about 481
Cisco onePK 482
Juniper, and netdev_stdlib 483, 484

native resources
reference link 230

NetApp filers
reference 491

Netcat 744

new Puppet 4 Master 161, 162

new template engine
leveraging 179, 180

NFS 522

Nginx
Phusion Passenger, using with 43-45

Nginx packages
reference link 44

node
about 224
classes, defining 314, 315
configuration files, defining 318
parameters, defining 315-317

node-less layout 314

nodeless setup 233

node migration
about 376, 377
advantages 376, 377

node-purge-ttl settings 291

node.rb ENC script
reference link 692

nodes
about 232
configuring, for ActiveMQ usage 698, 699, 700

nodes' classification 314

node-ttl settings 291

node update 377, 378

notable resource types
cron resource type 27
examining 22, 23
exec resource type 25, 26
mount resource type 27
user and group types 24

Nova module
references 349

official OpenStack module
about 360
URL 360
using 360

OLC (OpenLDAP configuration)
about 545
URL 545

old practices, breaking
about 183
dealing with bool algebra, on strings 184
dealing with different data types 188
hyphens, cleaning in names 186
reference syntax, learning 186
node inheritance, converting 183, 184
Ruby DSL anymore 187
relative class name resolution 187
strict variable naming, using 185

Open Container Initiative
URL 494

OpenLDAP configuration
defining 545-550

open source license 222

OpenStack component 359

OpenStack example
about 359
component (application) modules 359, 360
official OpenStack module, using 360
OpenStack infrastructure,
 puppetizing 363, 364
Puppet Labs OpenStack
 module, using 361, 363

OpenStack modules
URL 349

operating system 388

order management
about 245, 246
run stages 247

order of evaluation, controlling
about 12, 13
circular dependencies, avoiding 18, 19, 20
dependencies, declaring 14, 15, 16, 17
error propagation 17

O

obsolete exports
removing 150

P

package management 500

Packer
URL 494

PagerDuty service
 URL 470

PanoPuppet
 URL 693

parameterized classes
 about 659, 660
 caveats 93, 94

parameters
 application 316
 component 316
 country 316
 datacenter 316
 defining, for each node 315-317
 entry points 343
 env 315
 file-based configurations 343
 naming standards 344
 project 316
 role 315
 setting-based configurations 343
 site 316
 tenant 316
 using 342, 343
 zone 316

params pattern 337, 338

parser functions 98, 176

Passenger
 about 424
 comparing, with puppetserver 45, 46
 configuring 425-428
 installing 425-428

Pathogen
 installing 401
 URL 401

performance bottlenecks
 avoiding, from templates 139

performance considerations 42

performance dashboard 294, 295

performance dashboard, metrics
 Catalog duplication 295
 Command processing 295
 Command queue 295
 Discarded 296
 JVM Heap memory usage 295
 Nodes 295
 Processed 295

Rejected 296
Resources 295
Retried 295

performance implications, of container relationships 90

performance measurement
 about 441
 code optimization 443, 444
 Puppet Metrics 442
 Puppet versions, testing 445

personal branch 753

Phusion Passenger
 using, with Nginx 43, 44, 45

ping 744

plugins
 about 633
 agent, enhancing through 118, 119

pluginsync
 about 451, 452
 for Ruby facts distribution 460, 461, 462

PostgreSQL
 configuring 712
 installing 712
 reference link 290

postgresql module
 URL 464

profiles
 about 311-322
 managing 322, 323

Profiles pattern
 implementing 214-216

profiling 755, 756

properties 6, 7

providers
 about 672
 implementing 63, 64
 model, substantiating with 61, 62
 summarizing 62

proxy mode, Puppet
 SNMP 478
 Telnet or SSH connections 478
 Web API 478

proxy mode, with puppet device application
 about 479, 480
 reference 479

public modules
about 313
GitHub, using for 599-601

Puppet
about 507
and Docker 491-494
components 224, 225
configuring 225-227
configuring, for PuppetDB usage 714
executing 228, 229
Foreman, attaching to 689-691
installing 4, 5, 225-227, 711
master machine, setting up 31, 32
master manifest, creating 32, 33
native mode 478
on storage devices 491
proxy mode 478
scaling 422
URL, for installation 226

Puppet 3.8
using 161

Puppet 3 automatic
parameter lookup 271, 272

Puppet 3 functions API
limitations 176

Puppet 4
upgrading to 160

Puppet 4 functions
creating 176-178

Puppet 4.x
future enhancement 501

Puppet 5 496

Puppet agent
certificate, renewing 40
life cycle 38, 39
running, from cron 41
setting up 34-37
updating 163

puppet apply command
output, interpreting of 7, 8

Puppet architecture
about 312, 313
classes, defining 319
classes, defining for each node 314, 315
components 312, 313
custom resources, defining 319

parameters, defining for
each node 315, 316, 317
tasks 312

Puppet architecture, examples
about 324
default approach 325, 326, 327
ENC, used with Hiera backend 328
facts driven approach 332, 333
Foreman and Hiera, using 329
Foreman smart variables, using 331
Hiera-based setup 330
nodeless site.pp 333, 334

Puppet beaker
about 743
reference link 743

Puppetboard

about 223, 296
URL 296, 693

Puppet certificate authority
splitting off, from load
balancing machine 523

Puppet changes
propagating 417

Puppet code
deploying 414
deploying, with librarian-puppet 415
deploying, with r10k 416
modifications 309, 310
testing 407
testing, with Beaker 412-414
testing, with beaker-rspec 412-414
testing, with rspec-puppet 408-410
testing, with Vagrant 410, 411
writing 399, 400
writing, in Geppetto 400
writing, in Vim 401

Puppet code, stages
configuration 448
instantiation 448
parsing and compiling 448
report 448
transport 448

Puppet collections
about 225
reference link 509

Puppet Conf
reference link 502

puppet.conf file, parameters
ca_ttl 432
dns_alt_names 432

Puppet Dashboard 223, 313

PuppetDB
about 223, 707
configuration 284-287
configuring 707-710
configuring, for PostgreSQL usage 713
installation 284-287
installing 711
installing, manually 711
reference link 707
stack, completing with 46
URL, for installation 285

PuppetDB API
about 296
commands, using 299, 300
queries, using 296, 297, 298, 299

puppetdb module
reference link 293
URL 287

puppetdbquery module
about 306
Hiera backend 308, 309
query format 307
querying, from command line 307
querying, from Puppet manifests 308

puppetdb-termini package
installing 293

Puppet DSL
code, testing 163
user variables 240, 241

Puppet ecosystem
about 222, 223
configuration
management, importance 223, 224

Puppet Enterprise 222 223

Puppet Explorer
URL 693

Puppet extension
about 450
ENC, extendibility 450
Hiera, extendibility 450

indirector 452-456
pluginsync 451, 452

Puppet, for cloud and virtualization
about 484
Amazon Web Services 487
AWS provisioning 489
Cloud provisioning 490
VMware 485

Puppet Forge 358

Puppet-friendly software, characteristics
client-side registration 396
compassable configuration files, using 395
disabled services, installing 396
online configuration reload 396
standard packaging system, using 395

Puppet GUIs 693

Puppet Labs
URL, for documentation 43, 713

Puppet Labs OpenStack module
URL 361
using 361, 363

**puppetlabs-release-
pc1-0.9.2-1.el7.noarch.rpm**
reference link 510

puppetlabs-strings module
reference link 101

Puppet Labs website
URL 570

puppet-lint
about 401
URL 407

Puppet manifests
puppetdbquery module, querying 308

Puppet master
about 29, 30, 535
based on Trapperkeeper 429, 430
building 509, 510
certificates, generating 511, 512
code consistent, keeping 520
configuration settings, inspecting 34
load balancer, using 513, 514
PuppetDB, configuring 293, 294
systemd, using 512
tasks 30
with Passenger 424

Puppet Metrics**442****Puppet module**

about 69

modules, creating with 639, 641

puppet-module-data

reference link 339

Puppet Modules Forge**335****Puppet, on network equipment**

about 477-479

cultural challenge 478

native Puppet, on network equipment 481

proxy mode, with puppet device application 479-481

technical challenge 478

Puppet OpenStack modules

URL 359

Puppet resource

used, for configuring cron 530, 531

Puppet run

anatomy 448-450

Puppet run times

distributing 438, 439

puppet.schema

reference 546

puppetserver

about 509

Passenger, comparing with 45, 46

reference link 510

tuning 43

Puppet server

about 30

dividing 507

puppetserver.conf file**227****Puppet-supported modules**

concat 612-617

firewall 623-627

inifile 618-623

logical volume manager (lvm) 627-629

standard library (stdlib) 630-632

using 612

puppet-sync

about 585-587

URL 585

Puppet versions

testing 445

Q**queries**

endpoints 297

using 297-299

query parameters**297****R****r10k**

about 606

Puppet code, deploying with 416

reference link 606

using 606-611

rake**RBCommons**

URL 407

RedHat**442****redundancy**

saving, resource defaults 153, 154

relationships

establishing, among containers 84

reporting

about 508

turning on 681, 682

reporting, Puppet Labs

reference link 681

report types, Puppet

reference link 681

reserved names

about 248

reference link 248

Resource Abstraction**Layer (RAL)** 230, 231, 448**resource declaration**

attributes 229

title 229

type 229

resource defaults

about 237

used, for saving redundancy 153, 154

resource interaction

implementing 20, 21, 22

resource multiplexers

defined types, using as 79

resource parameters
 overriding 151, 152

resource references 237

resources
 about 6, 7, 229, 230
 converting, to data 207-210
 exporting 145
 exporting, to agents 144
 importing 145
 realizing, collectors used 143, 144

resource tags 718

resource type
 life cycle, on agent side 59, 60

resource types
 implementing 63, 64
 summarizing 62
 using, with generic providers 62

resource wrappers
 defined types, using as 77, 78

REST endpoints
 about 300
 /aggregated-event-counts endpoint 306
 /catalogs endpoint 303
 /event-counts endpoint 305
 /events endpoint 304
 /fact-names endpoint 303
 /facts endpoint 301
 /metrics endpoint 303
 /nodes endpoint 302
 /reports endpoint 304
 /resources endpoint 301
 /server-time endpoint 306
 /version endpoint 306

reusability patterns
 about 345
 files, managing 346, 348

reusable modules
 characteristics 345

roles 311, 321-323

roles and profiles pattern 357

Roles pattern
 implementing 214-216

rpm
 creating 528, 530

rspec-puppet
 about 401, 743
 reference link 743

 URL 408
 using 408-410

rspec-puppet tool
 reference link 107

rsync
 about 520, 521
 using 522

Ruby 222

Ruby facts 460, 462

Ruby version
 URL 386

Ruby Version Manager (RVM)
 URL 418

run stages 247

S

scope 238, 754, 755

searching 707

selectors 248, 337

SELinux
 about 517, 518
 reference link 517

separate data storage
 need for 192

serialization format
 modifying 496, 497

settings-based OpenStack's official module 363

shared modules
 about 317
 configuration files, placing 318

simple values
 working with 200, 201

singletons 138

site manifest 32

site module(s)
 about 313, 315, 318
 configuration files, placing 318

slapd
 reference 545

smart variables 331

Software Collections (SCL)
 URL 688

Springdale
reference link 509
SSH host keys
exporting 147
sshkey collection, for laptops 720-722
SSH keys
creating 522
SSL issues
troubleshooting 46, 47
SSL termination
managing 433, 434, 435
stack
completing, with PuppetDB 46
stack modules
reusability 364-369
stage resource type 247
standard de facto pattern
drawbacks 340
pros and cons 340
standardize naming conventions 344
standard library (stdlib) 630-632
Start of Authority (SOA) 730
states 245
static files
using 346
static scoping 244
stdlib
referernce link 630
stdlib module 104 341
stdmod
URL 345
stdmod shared module
URL 349
storeconfigs 508, 707
store configs code
reference link 440
stored configs
scaling 439-441
store mechanism
enabling 682, 683
structured design patterns 73
structured facts 57
sub-CDN role
creating 738-740
subqueries
building 297

summarizing 755, 756
systemd
reference link 512
using 512
system integrations 494
system knowledge
enhancing, through facts 127
systems
summarizing, with Facter 49-51
System Security Services
Daemon (SSSD) 618
system's facts
about 238
System Under Test (SUT)
about 412

T

tasks 311
templates
about 637
performance bottlenecks, avoiding from 139
using 346
using, in practice 137, 138
variables, using in 137
template syntax 136, 137
termini 452
Foreman 223
Tiny Puppet
about 369, 370
tp**conf 371
tp**dir 371
tp**install 371
tp**repo 371
URL 369
TLS headers 517
top scope variables 316
traffic compression 437
transactions 58
Trapperkeeper
about 429, 430
reference link 509
Travis
about 418, 419
URL 418

type-checking 241, 242

types

about 672

creating 673-679

type system

about 58

using 164-172

U

update path, Puppet agent

reference link 163

user and group types 24

user defined types 235

user-defined variables 224

users

managing 352, 353

user variables

in ENC 242

in Hiera 242

in Puppet DSL 240, 241

utilities, creating for derived manifests

about 112

configuration items, adding 113, 114

customization, allowing 115

dealing, with complexity 117, 118

unwanted configuration

items, removing 116, 117

V

Vagrant

URL 410

using 410, 411

vagrant commands

vagrant destroy [machine] 411

vagrant halt [machine] 411

vagrant provision [machine] 411

vagrant ssh <machine> 411

vagrant status 411

vagrant up [machine] 411

variable

about 238, 329

using 11

using, in templates 137

variable scope

about 244, 245

class scope 244

node scope 244

sub class scope 244

top scope 244

variable types 11, 12

Varnish

reference link 738

vCenter 485

Vim 401

Vim bundles

Syntastic 401

vim-puppet 401

vim-puppet

about 401

URL 401

Virtual Machines (VM) 410

virtual resources

about 252, 644

creating 139, 141

VMware

about 485

vCenter configuration 485

VM provisioning, on vCenter

and vSphere 485

vSphere virtual machines management,

with resource types 486, 487

W

Webrick 424

webserver.conf file 227

wget 745

workload

splitting 524-526

X

X-Client-DN 434

X-Client-Verify 434

X-SSL-Subject 434

Y

YAML

URL 537

yum repository

creating 531, 532



Thank you for buying
**Puppet: Mastering Infrastructure
Automation**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

