Community Experience Distilled

Mastering Puppet

Second Edition

Master Puppet for configuration management of your systems in an enterprise deployment

Thomas Uphill

[PACKT] open source*
PUBLISHING
community experience distilled

Mastering Puppet

Second Edition

Master Puppet for configuration management of your systems in an enterprise deployment

Thomas Uphill

[PACKT] open source
PUBLISHING
community experience distilled

BIRMINGHAM - MUMBAI

Mastering Puppet

Second Edition

First published: July 2014

Second edition: February 2016

Production reference: 1220216

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

ISBN 978-1-78588-810-6

www.packtpub.com

Credits

Author
Thomas Uphill

Reviewer
Bas Grolleman

Commissioning Editor
Priya Singh

Acquisition Editor
Nadeem Bagban

Content Development Editor
Mehvash Fatima

Technical Editor
Taabish Khan

Copy Editors
Ting Baker
Sneha Singh

Project Coordinator
Kinjal Bari

Proofreader
Safis Editing

Indexer
Tejal Daruwale Soni

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Author

Thomas Uphill is a long-time user of Puppet. He has presented Puppet tutorials at LOPSA-East, Cascada, and PuppetConf. He has also been a system administrator for over 20 years, working primarily with RedHat systems; he is currently a RedHat Certified Architect (RHCA). When not running the Puppet User Group of Seattle (PUGS), he volunteers for the LOPSA board and his local LOPSA chapter, SASAG. He blogs at `http://ramblings.narrabilis.com`.

About the Reviewer

Bas Grolleman works as a self-taught freelance Puppet professional in the Netherlands, he has his name in the code of many large-scale deployments. He learned the pain of scaling Puppet the hard way, that is, trial and error and spending hours going through a maze of dependencies. Now, he mostly tells people to take the time to do it right.

www.PacktPub.com

eBooks, discount offers, and more

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

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

PACKTLIB™

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

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

Why subscribe?

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

Table of Contents

Preface

The complexity of your installation will increase with the number of nodes in your organization. Working on a small deployment with a few developers is much simpler than working on a large installation with many developers.

Mastering Puppet Second Edition deals with the issues faced by larger deployments, such as scaling and versioning. This book will show you how to fit Puppet into your organization and keep everyone working. The concepts presented can be adopted to suit any size organization.

What this book covers

Chapter 1, *Dealing with Load/Scale*, will show you how to scale your Puppet infrastructure as your node count increases.

Chapter 2, *Organizing Your Nodes and Data*, is where we show different methods of applying modules to nodes. We look at Hiera and external node classifiers (ENCs).

Chapter 3, *Git and Environments*, introduces Git and how to use Git as an integral component of your Puppet infrastructure.

Chapter 4, *Public Modules*, shows how to use Puppet Forge as a source of modules and how to use several popular modules in your organization.

Chapter 5, *Custom Facts and Modules*, is where we extend Puppet with custom facts and start writing our own modules.

Chapter 6, *Custom Types*, is where we introduce defined types and show how to extend the Puppet language with our own custom types and providers.

Chapter 7, *Reporting and Orchestration*, is where we configure reporting to help track down errors in our Puppet deployments.

Chapter 8, *Exported Resources*, explains how useful it is to have resources on one node that can be applied to other nodes in your organization.

Chapter 9, *Roles and Profiles*, shows a popular design pattern in Puppet node deployments. Here we present the concept and show example usage.

Chapter 10, *Troubleshooting*, is where we show some common errors found in Puppet deployments, as well as possible solutions.

What you need for this book

All the examples in this book were written and tested using an Enterprise Linux 7 derived installation, such as CentOS 7, Scientific Linux 7, or Springdale Linux 7. Additional repositories used were EPEL (Extra Packages for Enterprise Linux), the Software Collections (SCL) repository, the Foreman repository, and the Puppet Labs repository. The version of Puppet used was the latest 4.2 series at the time of writing.

Who this book is for

This book is for those who have intermediate knowledge of Puppet and are looking to deploy it in their environment. Some idea of how to write simple modules for configuration management with Puppet is a prerequisite for this book.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Now sign the certificate using the `puppet cert sign` command."

A block of code is set as follows:

```
yumrepo { 'example.com-puppet':
  baseurl  => 'http://puppet.example.com/noarch',
  descr    => 'example.com Puppet Code Repository',
  enabled  => '1',
  gpgcheck => '0',
}
```

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

```
node_terminus = ldap
ldapserver = ldap.example.com
ldapbase = ou=hosts,dc=example,dc=com
```

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

```
# puppetserver gem install jruby-ldap
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "You can also navigate to the **Monitor | Reports** section to see the latest reports."

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

> Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

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

Errata

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

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

Piracy

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

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

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

Questions

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

1
Dealing with Load/Scale

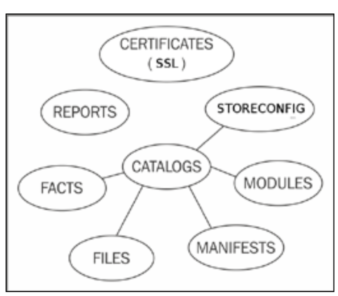
A large deployment will have a large number of nodes. If you are growing your installation from scratch, you might have to start with a single Puppet master. At a certain point in your deployment, a single Puppet master just won't cut it—the load will become too great. In my experience, this limit is 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::CertificateRequestpuppet.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-----
MIIFvDCCA6SgAwIBAgIBAjANBgkqhkiG9w0BAQsFADAoMSYwJAYDVQQDDB1QdXBw
```

```
...
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 `lb` 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 ~]$ sudosystemctl disable firewalld.service
rm '/etc/systemd/system/dbus-org.fedoraproject.FirewallD1.service'
rm '/etc/systemd/system/basic.target.wants/firewalld.service'
[thomas@client ~]$ sudosystemctl 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 `lb` machine and two machines running `puppetserver` and acting as Puppet masters. I will name these `puppetmaster1` and `puppetmaster2`.

To configure the `lb` 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>
  ServerNamepuppet.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 `lb`. 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 `lb` machine; name these servers, as `puppetmaster1` and `puppetmaster2`.

In our load balancing configuration, communication between the `lb` machine and the Puppet masters will be unencrypted. To maintain security, a private network should be established between the `lb` 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 `lb` 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 `lb`. 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:C9:0
E: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 `lb` 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 `lb` 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 `lb` machine, all the certificate traffic is directed to the `puppetserver` process running locally on the `lb` 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:B9: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 %>\
nManaged 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=c
om"
```

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/manifest/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 '{md
5}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/ogv/Git_For_Ages_4_And_Up.ogv`.

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 control2]$ 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-file20151001-12501-1y5tl02      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-syn
c/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`
- `inifile`
- `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 `order05`, 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` 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, `el6`, 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.

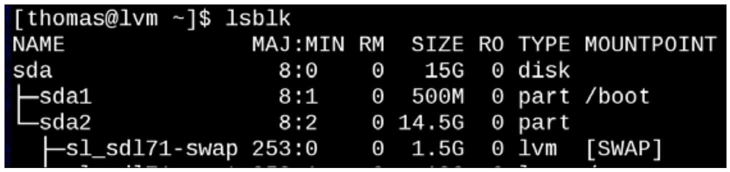
Logical volume manager

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:



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",
  }
}
```

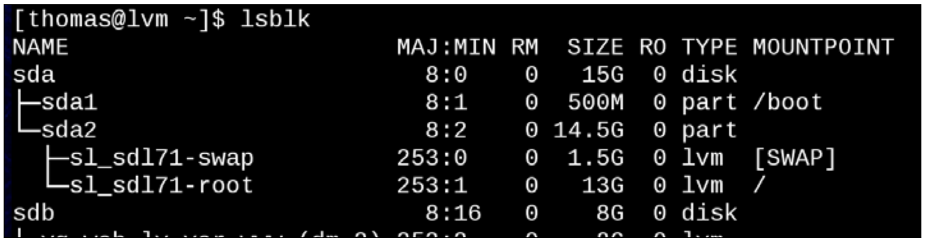
Now we'll create an `lvm.yaml` file in `hieradata/hosts/lvm.yaml`:

```
---
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:

```
[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
```

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:

```
module-name
├── files
├── lib
│   ├── augeas
│   │   └── lenses
│   ├── facter
│   └── puppet
│       ├── parser
│       │   └── functions
│       ├── provider
│       │   └── provider-name
│       │       (exec, file, package, etc...)
│       └── type
├── manifests
│   └── init.pp
└── templates
```

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/templating.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` 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.rbexample_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
      ifsubnet.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:

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:

```
classlvm_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',
  }
}
```

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,
  forcelocal => $forcelocal,
  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']]
  }
```

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]/Fi
le[/opt/puppetlabs/puppet/cache/concat/_var_home_nate_.bashrc/fragments/10_bashrc_user_nate]: S
cheduling 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 E
xec[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: crea
ted
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/fragments]: Scheduling re
fresh 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]/ensur
e: 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]/e
nsure: created
Notice: /Stage[main]/Main/Node[admins]/Example::Admin[nate]/Concat::Fragment[bash_profile_heade
r_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_p
rofile_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}f939eb71a81a9da364410b799
e817202'
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
```

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
```

```
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 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.yaml201509160551.yaml201509252128.yaml201510031025.
yaml201510040502.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::SSL::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/0mzOq/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 Revokation 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'
```

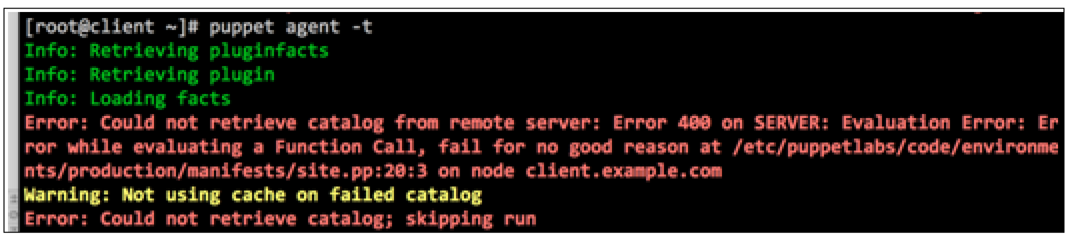
We are almost ready; the IRC report plugin uses the `carrier-pigeon` Ruby gem to do the IRC work, so we'll need to install that now. Since reports run within the `puppetserver` process, we need to install the gem within `puppetserver`, as shown here:

```
[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
```

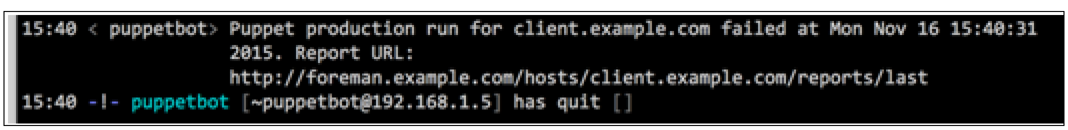
Now we can restart `puppetserver` on our `reports` worker and create a catalog compilation problem on the client. To ensure the catalog fails to compile, I've edited `site.pp` and added the following line to the default node definition:

```
fail('fail for no good reason')
```

This causes the catalog to fail compilation on our client node as shown in the following screenshot:



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:



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
```

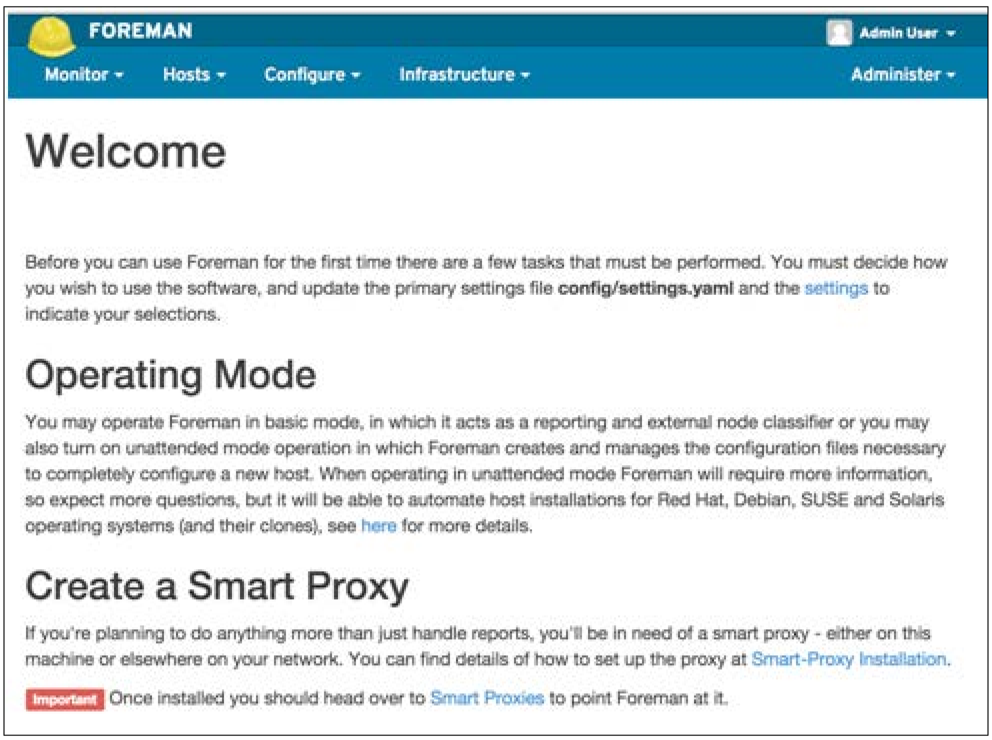
The `foreman-installer` command uses `puppet apply` to configure Foreman on the server. Since we will only be using Foreman for reporting in this example, we can just use the installer, as shown here:

```
[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:

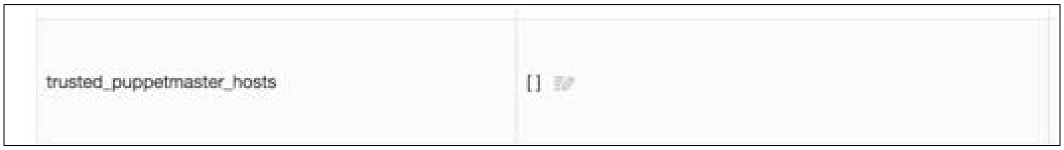
```
---
# Update for your Foreman and Puppet master hostname(s)
:url: "https://foreman.example.com"
:ssl_ca: "/etc/puppetlabs/puppet/ssl/certs/ca.pem"
:ssl_cert: "/etc/puppetlabs/puppet/ssl/certs/reports.example.com.pem"
:ssl_key: "/etc/puppetlabs/puppet/ssl/private_keys/reports.example.
com.pem"

# 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:

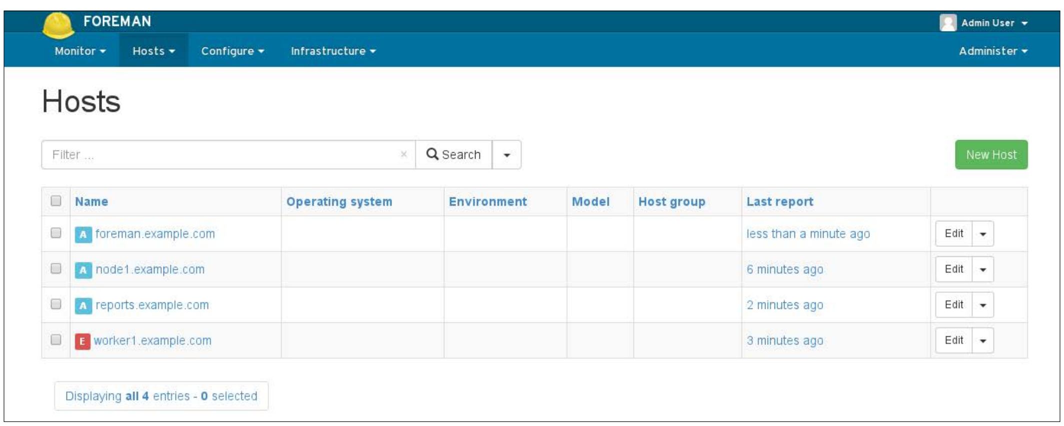


Note that this must be an array, so keep the `[ ]` brackets around `reports.example.com`, as shown in the following screenshot:

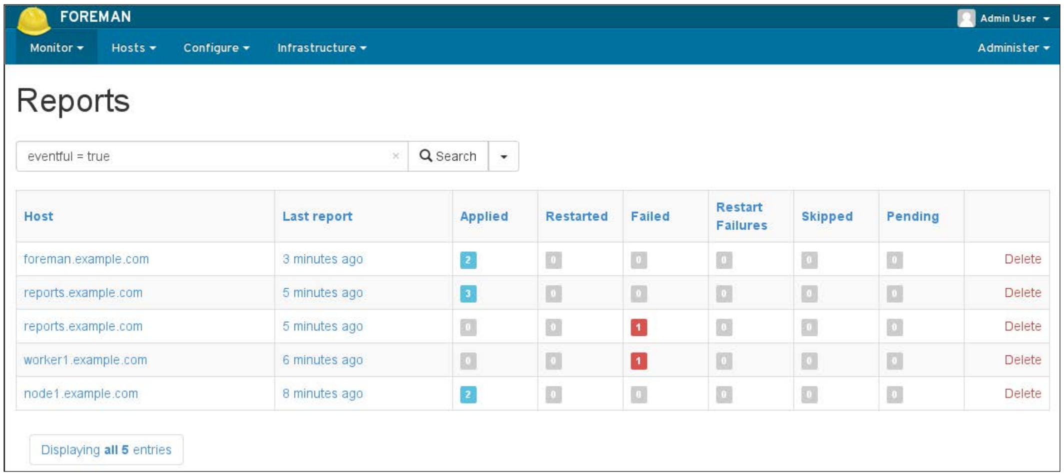
With all this in place, when a node compiles a catalog, it will send the report to `reports.example.com`, which will send the report on to `foreman.example.com`. After a few reports arrive, our Foreman homepage will list hosts and reports.

Using Foreman

Let's first look at the **Hosts** window shown in the following screenshot:



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:



Clicking on `client.example.com` shows the failed catalog run and the contents of the error message, as shown in the following screenshot:

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

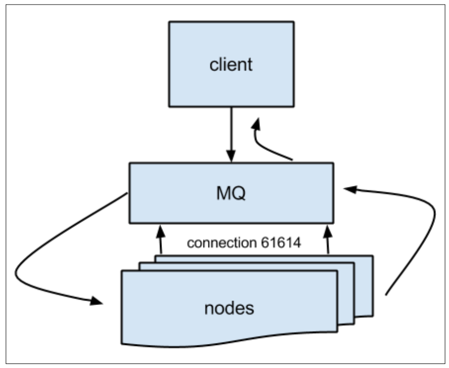
Representing Puppet report information in a web GUI is a useful idea. There are several GUIs available; Puppet Labs has Puppet Enterprise and its console interface. Other open source alternatives are **Puppetboard** (`https://github.com/voxpupuli/puppetboard`), **PanoPuppet** (`https://github.com/propyless/panopuppet`), 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]# mvactivemq.xmlactivemq.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,everyo
ne"/>
</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.pempuppet.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
```

8. 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]# catpuppet.example.com.pempuppet.example.com.
private.pem>puppet.pem
[root@standtmp]# opensslpkcs12 -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
```

9. 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`.

10. 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
```

11. 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::SSL::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-----
MIIFcTCCA1mgAwIBAgIBGjANBgkqhkiG9w0BAQsFADAoMSYwJAYDVQQDDB1QdXBw
…
-----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-----
MIIFWzCCA0OgAwIBAgIBEzANBgkqhkiG9w0BAQsFADAoMSYwJAYDVQQDDB1QdXBw
...
Vd5M0lfdYSDKOA+b1AXXoMaAn9n9j7AyBhQhie52Og==
-----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-----
MIIFfjCCA2agAwIBAgIBATANBgkqhkiG9w0BAQsFADAoMSYwJAYDVQQDDB1QdXBw


...
XO+dgA5aAhUUMg==
-----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.

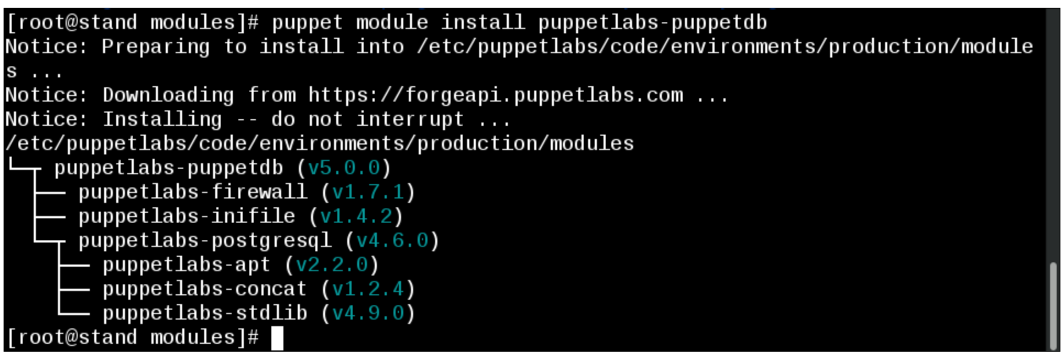
Configuring PuppetDB – using the Forge module

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:



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-pc1-el-7.noarch.rpm

puppetlabs-release-pc1-el-7.noarch.rpm                         | 5.1 kB 00:00:00

Examining /var/tmp/yum-root-dfBZAN/puppetlabs-release-pc1-el-7.noarch.rpm: puppetlabs-release-pc1-1.0.0-1.el6.noarch

Marking /var/tmp/yum-root-dfBZAN/puppetlabs-release-pc1-el-7.noarch.rpm as an update to puppetlabs-release-pc1-0.9.2-1.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',
}
```

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::CertificateRequestpuppetdb-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` node 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 localhostpuppetdbpuppetdb
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:

```
classldap::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 ~]# grepldap /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:0
C: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-rsaAAAAbd3dz56c2E...
```

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, `clustername`, that defines the name of the cluster based on information either on the node or in a central **Configuration Management Database** (**CMDB**).

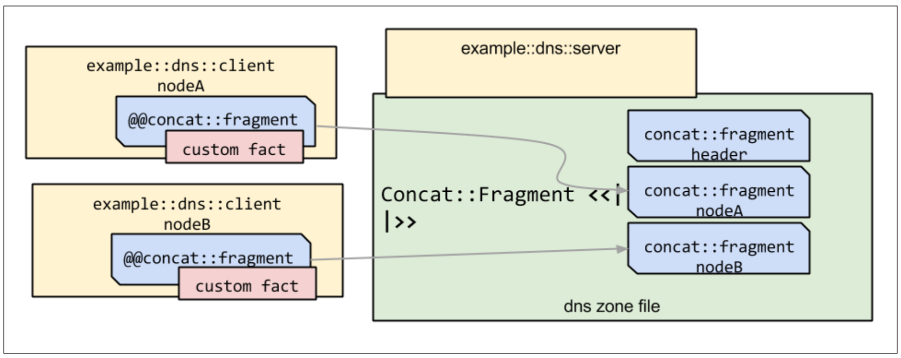
> CMDBs are the data warehouses of an organization. Examples of CMDBs include OneCMDB, Itop, or BMC Atrium.

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=="clustername"`. 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.

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'
    ) {
```

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\ndomain $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]+)$','\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']
  }
```

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.

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

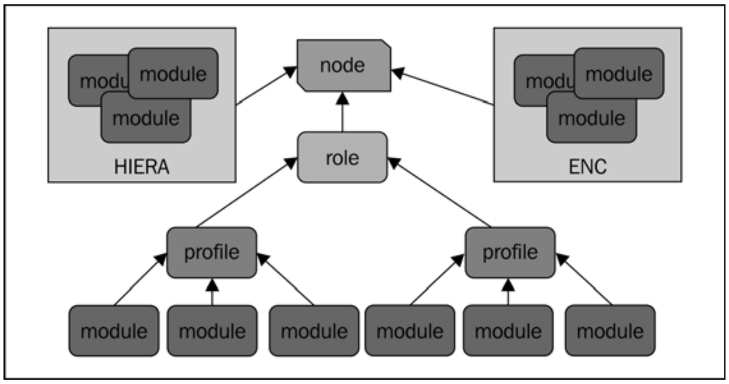
Roles and Profiles

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
ZXQgQ0E6IHB1cHBldC5leGFtcGxlLmNvbTAeFw0xNTA5MTAwNTI2MTVaFw0yMDA5
MDkwNTI2MTVaMCgxJjAkBgNVBAMMHVB1cHBldCBDQTogcHVwcGV0LmV4YW1wbGUu
Y29tMIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAh4TW/ge8Pqj//EA1
c8UePnh2XyG0w3UAeF1o7ptGBEdmYJinG+P7QgbDk44ySzVqyI3WpD1I22qZGXnS
DYhNq7NsKnNgJftGE4MpHYiqBzuDtk0g0SqnGFe01YYBNPFEiPcA6RGPi3YNfe1P
Fd8wlkMubjLfIoFtDG6AZHVW9m08j9gA1UOPDrfgKy9Ye71oL6DYRMaUp9MpFAq0
tQr+uoAH/LS5EXam7+DRk0c1MCRddE8OUmtR8RjsFSKMguQL2C0gw5hLNCLDEcox
```

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
```

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-----
MIIFfjCCA2agAwIBAgIBATANBgkqhkiG9w0BAQsFADAoMSYwJAYDVQQDDB1QdXBw
ZXQgQ0E6IHB1cHBldC5leGFtcGxlLmNvbTAeFw0xNTA5MTAwNTI2MTVaFw0yMDA5
MDkwNTI2MTVaMCgxJjAkBgNVBAMMHVB1cHBldCBDQTogcHVwcGV0LmV4YW1wbGUu
Y29tMIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAh4TW/ge8Pqj//EA1
c8UePnh2XyG0w3UAeF1o7ptGBEdmYJinG+P7QgbDk44ySzVqyI3WpD1I22qZGXnS
DYhNq7NsKnNgJftGE4MpHYiqBzuDtk0g0SqnGFe01YYBNPFEiPcA6RGPi3YNfe1P
Fd8wlkMubjLfIoFtDG6AZHVW9m08j9gA1UOPDrfgKy9Ye71oL6DYRMaUp9MpFAqO
tQr+uoAH/LS5EXam7+DRk0c1MCRddE8OUmtR8RjsFSKMguQL2C0gw5hLNCLDEcox
```

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-----
MIIFfjCCA2agAwIBAgIBATANBgkqhkiG9w0BAQsFADAoMSYwJAYDVQQDDB1QdXBw
ZXQgQ0E6IHB1cHBldC5leGFtcGxlLmNvbTAeFw0xNTA5MTAwNTI2MTVaFw0yMDA5
```

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',
  }
}
```

The file resource is defined twice with the same name. The error appears when we run Puppet, as shown in the following screenshot:

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 sec
onds
{
  "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
}
```

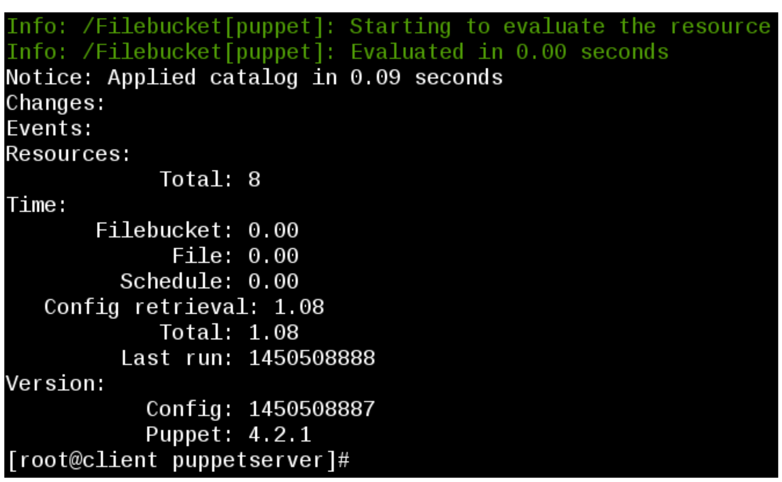
If we run `puppet agent` against this version, we see the notification we were expecting, as follows:

```
# 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.

Index

I

J

L

M

N

O

P

R

S

T

V

W

Y

Thank you for buying

Mastering Puppet

Second Edition

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`.

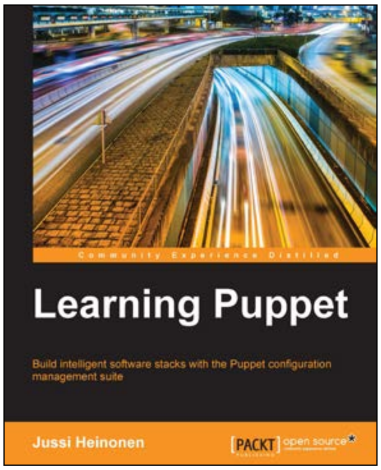
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, 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.

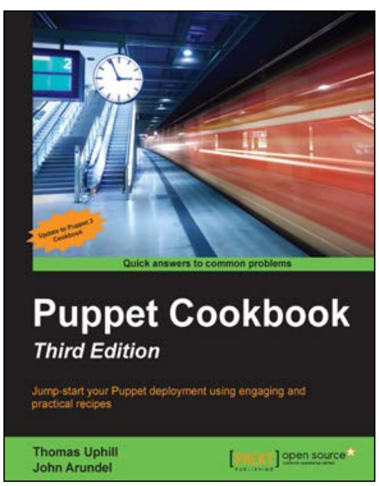


Learning Puppet

ISBN: 978-1-78439-983-2 Paperback: 304 pages

Build intelligent software stacks with the Puppet configuration management suite

1. Extends your skills beyond the built-in functionalities of Puppet, in a stepwise manner.
2. This seasoned systems developer will give you a plethora of knowledge to build intelligent software stacks, with tips and tricks.
3. Helps troubleshooting commonly occurring problems.



Puppet Cookbook

Third Edition

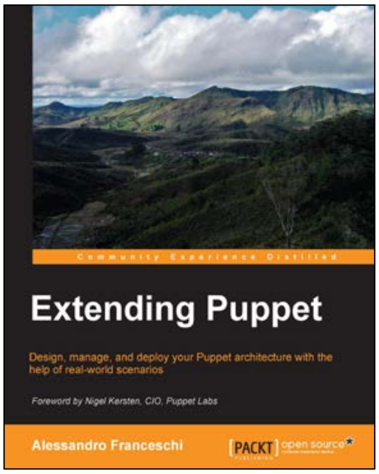
ISBN: 978-1-78439-488-2 Paperback: 336 pages

Jump-start your Puppet deployment using engaging and practical recipes

1. Engaging and practical recipes with images.
2. This book covers distributed and centralized Puppet deployments.
3. The authors have worked on Puppet since the 0.24 version and thus are just the right people to teach the essentials of Puppet.

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

[PACKT] open source
community experience distilled
PUBLISHING

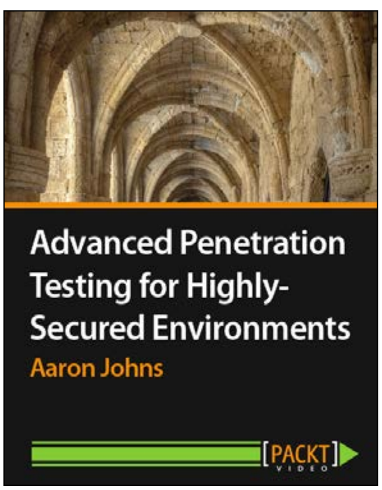


Extending Puppet

ISBN: 978-1-78398-144-1 Paperback: 328 pages

Design, manage, and deploy your Puppet architecture with the help of real-world scenarios

1. This book gives you the latest trends and best practices of extending Puppet.
2. The author is highly experienced in using Puppet and thus is excellent at teaching the under-the-hood concepts of Puppet.
3. Several examples of strategies and patterns of Puppet automation.



Advanced Penetration Testing for Highly-Secured Environments [Video]

ISBN: 978-1-78216-450-0 Duration: 02:50 hours

An intensive hands-on course to perform professional penetration testing

1. The video has hands on examples and is task-based.
2. This video covers all security tools.
3. The author is highly experienced in business networks and security policies and teaches under-the-hood concepts of penetration testing.

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