Thomas Uphill, John Arundel
Neependra Khare, Hideto Saito,
Hui-Chuan Chloe Lee, Ke-Jou Carol Hsu

DevOps: Puppet, Docker, and Kubernetes

Learning Path

Get hands-on recipes to automate and manage Linux containers with the Docker 1.6 environment and jump-start your Puppet development

Packt

DevOps: Puppet, Docker, and Kubernetes

Get hands-on recipes to automate and manage Linux containers with the Docker 1.6 environment and jump-start your Puppet development

A course in three modules

BIRMINGHAM - MUMBAI

DevOps: Puppet, Docker, and Kubernetes

Published on: March 2017

Production reference: 1150317

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

ISBN: 978-1-78829-761-5

www.packtpub.com

Credits

Authors

Thomas Uphill

John Arundel

Neependra Khare

Hideto Saito

Hui-Chuan Chloe Lee

Ke-Jou Carol Hsu

Reviewers

Dhruv Ahuja

James Fryman

Jeroen Hooyberghs

Pedro Morgado

Scott Collier

Julien Duponchelle

Allan Espinosa

Vishnu Gopal

Matt Ma

Content Development Editor

Juliana Nair

Graphics

Kirk D'Penha

Production Coordinator

Shantanu N. Zagade

Preface

With so many IT management and DevOps tools on the market, both open source and commercial, it's difficult to know where to start. DevOps is incredibly powerful when implemented correctly, here's how to get it done.

What this learning path covers

Module 1, Puppet Cookbook (third edition), this module covers all aspects of your puppet infrastructure using simple easy to follow recipes that are independent and can be used to solve real world problems quickly.

Puppet Cookbook takes the reader from a basic knowledge of Puppet to a complete and expert understanding of Puppet's latest and most advanced features.
With emphasis on real-world implementation, this book delves into various aspects of writing good Puppet code, including using Puppet community style, checking your manifests with puppet-lint and community best practices. It then shows the readers how to set up Puppet for the first time, including instructions on installing Puppet, creating your first manifests, using version control with Puppet and so on. You will also learn to write better manifests, manage resources, files and applications. You'll then be introduced to powerful tools that have grown up around Puppet, including Hiera, Facter, and rspec-puppet. Finally, you will also learn to master common Monitoring, Reporting, and Troubleshooting techniques.

Updated with the latest advancements and best practices, this book gives you a clear view on how to "connect the dots" and expands your understanding to successfully use and extend Puppet.

Module 2, Docker Cookbook, this module aims to help you get working with Docker by providing you with step-by-step recipes that enable you to effectively deploy Docker in your development, test, and production environments.

You will start with verifying and installing Docker on different environments and look into understanding and working with containers and images. Next, you will move on to study the operations related to images. You then proceed to learn about network and data management for containers and how to build an environment for Continuous Integration with the help of services from companies like Shippable and Drone. The book then explores the RESTful APIs provided by Docker to perform different operations like image/container operations before taking a look at the Docker Remote API client. The book ends with a look at logs and troubleshooting Docker to solve issues and bottlenecks.

Module 3, Kubernetes Cookbook, Kubernetes is Google's solution to managing a cluster of containers. Kubernetes provides a declarative API to manage clusters while giving us a lot of flexibility. This book will provide you with recipes to better manage containers in different scenarios in production using Kubernetes.

We will start by giving you a quick brush up on how Kubernetes works with containers along with an overview of the main Kubernetes features such as Pods, Replication Controllers, and more. Next, we will teach you how to create Kubernetes cluster and how to run programs on Kubernetes. We'll explain features such as High Availability Kubernetes master setup, using Kubernetes with Docker, and orchestration with Kubernetes using AWS. Later, will show you how to use Kubernetes-UI, and how to set up and manage Kubernetes clusters on the cloud and bare metal.

Upon completion of this book, you will be able use Kubernetes in production and will have a better understanding of how to manage your containers using Kubernetes.

What you need for this learning path

The primary softwares required are as follows:

- Puppet 3.7.3
- Kubernetes 1.1.3Java
- Etcd 2.1.1
- Flanneld 0.5.2
- Docker 1.7.1
- Kubernetes 1.2.2
- Etcd 2.3.1
- Amazon Web Services

- entOS
  - 7.1/ubuntu
  - 14.04/Amazeon
  - Linux 2015.09
- Debian and Enterprise Linux-based distributions

Who this learning path is for

This Learning Path is for developers, system administrators, and DevOps engineers who want to use Puppet, Docker, and Kubernetes in their development, QA, or production environments. This Learning Path assumes experience with Linux administration and requires some experience with command-line usage and basic text file editing.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

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

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

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

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

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/DevOps-Puppet-Docker-and-Kubernetes`. We also have other code bundles from our rich catalog of books, videos, and courses available at `https://github.com/PacktPublishing/`. Check them out!

Errata

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

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

Piracy

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

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

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

Questions

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

Module 1: Puppet Cookbook

Module 2: Docker Cookbook

Module 1

Puppet Cookbook

Jump-start your puppet deployment using engaging and practical recipes

1
Puppet Language and Style

"Computer language design is just like a stroll in the park. Jurassic Park, that is."

— Larry Wall

In this chapter, we will cover the following recipes:

- Adding a resource to a node
- Using **Facter** to describe a node
- Installing a package before starting a service
- Installing, configuring, and starting a service
- Using community **Puppet** style
- Creating a manifest
- Checking your manifests with Puppet-lint
- Using modules
- Using standard naming conventions
- Using inline templates
- Iterating over multiple items
- Writing powerful conditional statements
- Using regular expressions in if statements
- Using selectors and case statements
- Using the in operator
- Using regular expression substitutions
- Using the future parser

Introduction

In this chapter, we'll start with the basics of Puppet syntax and show you how some of the syntactic sugar in Puppet is used. We'll then move on to how Puppet deals with dependencies and how to make Puppet do the work for you.

We'll look at how to organize and structure your code into modules following community conventions, so that other people will find it easy to read and maintain your code. I'll also show you some powerful features of Puppet language, which will let you write concise, yet expressive manifests.

Adding a resource to a node

This recipe will introduce the language and show you the basics of writing Puppet code. A beginner may wish to reference *Puppet 3: Beginner's Guide*, *John Arundel*, *Packt Publishing* in addition to this section. Puppet code files are called manifests; manifests declare resources. A resource in Puppet may be a type, class, or node. A type is something like a file or package or anything that has a type declared in the language. The current list of standard types is available on puppetlabs website at `https://docs.puppetlabs.com/references/latest/type.html`. I find myself referencing this site very often. You may define your own types, either using a mechanism, similar to a subroutine, named **defined types**, or you can extend the language using a custom type. Types are the heart of the language; they describe the things that make up a node (node is the word Puppet uses for client computers/devices). Puppet uses resources to describe the state of a node; for example, we will declare the following package resource for a node using a site manifest (`site.pp`).

How to do it...

Create a `site.pp` file and place the following code in it:

```
node default {
  package { 'httpd':
    ensure => 'installed'
  }
}
```

Downloading the example code

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

How it works...

This manifest will ensure that any node, on which this manifest is applied, will install a package called `'httpd'`. The `default` keyword is a wildcard to Puppet; it applies anything within the node default definition to any node. When Puppet applies the manifest to a node, it uses a **Resource Abstraction Layer** (**RAL**) to translate the package type into the package management system of the target node. What this means is that we can use the same manifest to install the `httpd` package on any system for which Puppet has a **Provider** for the package type. Providers are the pieces of code that do the real work of applying a manifest. When the previous code is applied to a node running on a YUM-based distribution, the YUM provider will be used to install the `httpd` RPM packages. When the same code is applied to a node running on an **APT**-based distribution, the APT provider will be used to install the `httpd` DEB package (which may not exist, most debian-based systems call this package `apache2`; we'll deal with this sort of naming problem later).

Using Facter to describe a node

Facter is a separate utility upon which Puppet depends. It is the system used by Puppet to gather information about the target system (node); `facter` calls the nuggets of information facts. You may run `facter` from the command line to obtain real-time information from the system.

How to do it...

1. Use `facter` to find the current uptime of the system, the uptime fact:

```
t@cookbook ~$ facter uptime
0:12 hours
```

2. Compare this with the output of the Linux uptime command:

```
t@cookbook ~$ uptime
 01:18:52 up 12 min,  1 user,  load average: 0.00, 0.00, 0.00
```

How it works...

When `facter` is installed (as a dependency for puppet), several fact definitions are installed by default. You can reference each of these facts by name from the command line.

There's more...

Running `facter` without any arguments causes `facter` to print all the facts known about the system. We will see in later chapters that `facter` can be extended with your own custom facts. All facts are available for you to use as variables; variables are discussed in the next section.

Variables

Variables in Puppet are marked with a dollar sign ($) character. When using variables within a manifest, it is preferred to enclose the variable within braces `"${myvariable}"` instead of `"$myvariable"`. All of the facts from `facter` can be referenced as top scope variables (we will discuss scope in the next section). For example, the **fully qualified domain name** (**FQDN**) of the node may be referenced by `"${::fqdn}"`. Variables can only contain alphabetic characters, numerals, and the underscore character (_). As a matter of style, variables should start with an alphabetic character. Never use dashes in variable names.

Scope

In the variable example explained in the *There's more...* section, the fully qualified domain name was referred to as `${::fqdn}` rather than `${fqdn}`; the double colons are how Puppet differentiates scope. The highest level scope, top scope or global, is referred to by two colons (`::`) at the beginning of a variable identifier. To reduce namespace collisions, always use fully scoped variable identifiers in your manifests. For a Unix user, think of top scope variables as the `/` (root) level. You can refer to variables using the double colon syntax similar to how you would refer to a directory by its full path. For the developer, you can think of top scope variables as global variables; however, unlike global variables, you must always refer to them with the double colon notation to guarantee that a local variable isn't obscuring the top scope variable.

Installing a package before starting a service

To show how ordering works, we'll create a manifest that installs `httpd` and then ensures the `httpd` package service is running.

How to do it...

1. We start by creating a manifest that defines the service:

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

2. The service definition references a package resource named `httpd`; we now need to define that resource:

```
package {'httpd':
  ensure => 'installed',
}
```

How it works...

In this example, the package will be installed before the service is started. Using `require` within the definition of the `httpd` service ensures that the package is installed first, regardless of the order within the manifest file.

Capitalization

Capitalization is important in Puppet. In our previous example, we created a package named `httpd`. If we wanted to refer to this package later, we would capitalize its type (`package`) as follows:

```
Package['httpd']
```

To refer to a class, for example, the `something::somewhere` class, which has already been included/defined in your manifest, you can reference it with the full path as follows:

```
Class['something::somewhere']
```

When you have a defined type, for example the following defined type:

```
example::thing {'one':}
```

The preceding resource may be referenced later as follows:

```
Example::Thing['one']
```

Knowing how to reference previously defined resources is necessary for the next section on metaparameters and ordering.

Learning metaparameters and ordering

All the manifests that will be used to define a node are compiled into a catalog. A catalog is the code that will be applied to configure a node. It is important to remember that manifests are not applied to nodes sequentially. There is no inherent order to the application of manifests. With this in mind, in the previous `httpd` example, what if we wanted to ensure that the `httpd` process started after the `httpd` package was installed?

We couldn't rely on the `httpd` service coming after the `httpd` package in the manifests. What we have to do is use metaparameters to tell Puppet the order in which we want resources applied to the node. Metaparameters are parameters that can be applied to any resource and are not specific to any one resource type. They are used for catalog compilation and as hints to Puppet but not to define anything about the resource to which they are attached. When dealing with ordering, there are four metaparameters used:

- `before`
- `require`
- `notify`
- `subscribe`

The `before` and `require` metaparameters specify a direct ordering; `notify` implies `before` and `subscribe` implies `require`. The `notify` metaparameter is only applicable to services; what notify does is tell a service to restart after the notifying resource has been applied to the node (this is most often a package or file resource). In the case of files, once the file is created on the node, a notify parameter will restart any services mentioned. The `subscribe` metaparameter has the same effect but is defined on the service; the service will subscribe to the file.

Trifecta

The relationship between package and service previously mentioned is an important and powerful paradigm of Puppet. Adding one more resource-type file into the fold, creates what puppeteers refer to as the **trifecta**. Almost all system administration tasks revolve around these three resource types. As a system administrator, you install a package, configure the package with files, and then start the service.

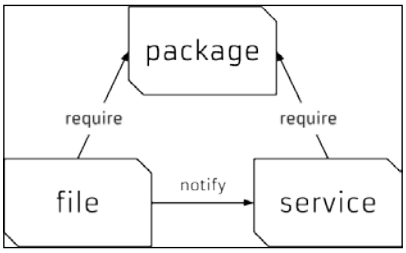

Diagram of Trifecta (Files require package for directory, service requires files and package)

Idempotency

A key concept of Puppet is that the state of the system when a catalog is applied to a node cannot affect the outcome of Puppet run. In other words, at the end of Puppet run (if the run was successful), the system will be in a known state and any further application of the catalog will result in a system that is in the same state. This property of Puppet is known as idempotency. **Idempotency** is the property that no matter how many times you do something, it remains in the same state as the first time you did it. For instance, if you had a light switch and you gave the instruction to turn it on, the light would turn on. If you gave the instruction again, the light would remain on.

Installing, configuring, and starting a service

There are many examples of this pattern online. In our simple example, we will create an Apache configuration file under `/etc/httpd/conf.d/cookbook.conf`. The `/etc/httpd/conf.d` directory will not exist until the `httpd` package is installed. After this file is created, we would want `httpd` to restart to notice the change; we can achieve this with a notify parameter.

How to do it...

We will need the same definitions as our last example; we need the package and service installed. We now need two more things. We need the configuration file and index page (`index.html`) created. For this, we follow these steps:

1. As in the previous example, we ensure the service is running and specify that the service requires the `httpd` package:

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

2. We then define the package as follows:

```
package {'httpd':
  ensure => installed,
}
```

3. Now, we create the `/etc/httpd/conf.d/cookbook.conf` configuration file; the `/etc/httpd/conf.d` directory will not exist until the `httpd` package is installed. The `require` metaparameter tells Puppet that this file requires the `httpd` package to be installed before it is created:

```
file {'/etc/httpd/conf.d/cookbook.conf':
  content => "<VirtualHost *:80>\nServername
    cookbook\nDocumentRoot
    /var/www/cookbook\n</VirtualHost>\n",
  require => Package['httpd'],
  notify => Service['httpd'],
}
```

4. We then go on to create an `index.html` file for our virtual host in `/var/www/cookbook`. This directory won't exist yet, so we need to create this as well, using the following code:

```
file {'/var/www/cookbook':
  ensure => directory,
}
file {'/var/www/cookbook/index.html':
  content => "<html><h1>Hello World!</h1></html>\n",
  require => File['/var/www/cookbook'],
}
```

How it works...

The `require` attribute to the file resources tell Puppet that we need the `/var/www/cookbook` directory created before we can create the `index.html` file. The important concept to remember is that we cannot assume anything about the target system (node). We need to define everything on which the target depends. Anytime you create a file in a manifest, you have to ensure that the directory containing that file exists. Anytime you specify that a service should be running, you have to ensure that the package providing that service is installed.

In this example, using metaparameters, we can be confident that no matter what state the node is in before running Puppet, after Puppet runs, the following will be true:

- `httpd` will be running
- The `VirtualHost` configuration file will exist
- `httpd` will restart and be aware of the `VirtualHost` file
- The `DocumentRoot` directory will exist
- An `index.html` file will exist in the `DocumentRoot` directory

Using community Puppet style

If other people need to read or maintain your manifests, or if you want to share code with the community, it's a good idea to follow the existing style conventions as closely as possible. These govern such aspects of your code as layout, spacing, quoting, alignment, and variable references, and the official puppetlabs recommendations on style are available at `http://docs.puppetlabs.com/guides/style_guide.html`.

How to do it...

In this section, I'll show you a few of the more important examples and how to make sure that your code is style compliant.

Indentation

Indent your manifests using two spaces (not tabs), as follows:

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

Quoting

Always quote your resource names, as follows:

```
package { 'exim4':
```

We cannot do this as follows though:

```
package { exim4:
```

Use single quotes for all strings, except when:

- The string contains variable references such as `"${::fqdn}"`
- The string contains character escape sequences such as `"\n"`

Consider the following code:

```
file { '/etc/motd':
  content => "Welcome to ${::fqdn}\n"
}
```

Puppet doesn't process variable references or escape sequences unless they're inside double quotes.

Always quote parameter values that are not reserved words in Puppet. For example, the following values are not reserved words:

```
name => 'Nucky Thompson',
mode => '0700',
owner => 'deploy',
```

However, these values are reserved words and therefore not quoted:

```
ensure => installed,
enable => true,
ensure => running,
```

False

There is only one thing in Puppet that is false, that is, the word `false` without any quotes. The string `"false"` evaluates to `true` and the string `"true"` also evaluates to true. Actually, everything besides the literal false evaluates to true (when treated as a Boolean):

```
if "false" {
  notify { 'True': }
}
if 'false' {
  notify { 'Also true': }
}
```

```
if false {
  notify { 'Not true': }
}
```

When this code is run through `puppet apply`, the first two notifies are triggered. The final notify is not triggered; it is the only one that evaluates to `false`.

Variables

Always include curly braces ({ }) around variable names when referring to them in strings, for example, as follows:

```
source => "puppet:///modules/webserver/${brand}.conf",
```

Otherwise, Puppet's parser has to guess which characters should be a part of the variable name and which belong to the surrounding string. Curly braces make it explicit.

Parameters

Always end lines that declare parameters with a comma, even if it is the last parameter:

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

This is allowed by Puppet, and makes it easier if you want to add parameters later, or reorder the existing parameters.

When declaring a resource with a single parameter, make the declaration all on one line and with no trailing comma, as shown in the following snippet:

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

Where there is more than one parameter, give each parameter its own line:

```
package { 'rake':
  ensure   => installed,
  provider => gem,
  require  => Package['rubygems'],
}
```

To make the code easier to read, line up the parameter arrows in line with the longest parameter, as follows:

```
file { "/var/www/${app}/shared/config/rvmrc":
  owner   => 'deploy',
  group   => 'deploy',
```

```
  content => template('rails/rvmrc.erb'),
  require => File["/var/www/${app}/shared/config"],
}
```

The arrows should be aligned per resource, but not across the whole file, otherwise it can make it difficult for you to cut and paste code from one file to another.

Symlinks

When declaring file resources which are symlinks, use `ensure => link` and set the target attribute, as follows:

```
file { '/etc/php5/cli/php.ini':
  ensure => link,
  target => '/etc/php.ini',
}
```

Creating a manifest

If you already have some Puppet code (known as a Puppet manifest), you can skip this section and go on to the next. If not, we'll see how to create and apply a simple manifest.

How to do it...

To create and apply a simple manifest, follow these steps:

1. First, install Puppet locally on your machine or create a virtual machine and install Puppet on that machine. For YUM-based systems, use `https://yum.puppetlabs.com/` and for APT-based systems, use `https://apt.puppetlabs.com/`. You may also use gem to install Puppet. For our examples, we'll install Puppet using gem on a Debian Wheezy system (hostname: `cookbook`). To use gem, we need the `rubygems` package as follows:

```
t@cookbook:~$ sudo apt-get install rubygems
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  rubygems
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 0 B/597 kB of archives.
After this operation, 3,844 kB of additional disk space will be
used.
```

```
Selecting previously unselected package rubygems.
(Reading database ... 30390 files and directories currently
installed.)
Unpacking rubygems (from .../rubygems_1.8.24-1_all.deb) ...
Processing triggers for man-db ...
Setting up rubygems (1.8.24-1) ...
```

2. Now, use `gem` to install Puppet:

```
t@cookbook $ sudo gem install puppet
Successfully installed hiera-1.3.4
Fetching: facter-2.3.0.gem (100%)
Successfully installed facter-2.3.0
Fetching: puppet-3.7.3.gem (100%)
Successfully installed puppet-3.7.3
Installing ri documentation for hiera-1.3.4
Installing ri documentation for facter-2.3.0
Installing ri documentation for puppet-3.7.3
Done installing documentation for hiera, facter, puppet after 239
seconds
```

3. Three gems are installed. Now, with Puppet installed, we can create a directory to contain our Puppet code:

```
t@cookbook:~$ mkdir -p .puppet/manifests
t@cookbook:~$ cd .puppet/manifests
t@cookbook:~/.puppet/manifests$
```

4. Within your `manifests` directory, create the `site.pp` file with the following content:

```
node default {
  file { '/tmp/hello':
    content => "Hello, world!\n",
  }
}
```

5. Test your manifest with the `puppet apply` command. This will tell Puppet to read the manifest, compare it to the state of the machine, and make any necessary changes to that state:

```
t@cookbook:~/.puppet/manifests$ puppet apply site.pp
Notice: Compiled catalog for cookbook in environment production in
0.14 seconds
Notice: /Stage[main]/Main/Node[default]/File[/tmp/hello]/ensure:
defined content as '{md5}746308829575e17c3331bbcb00c0898b'
Notice: Finished catalog run in 0.04 seconds
```

6. To see if Puppet did what we expected (create the `/tmp/hello` file with the `Hello, world!` content), run the following command:

```
t@cookbook:~/puppet/manifests$ cat /tmp/hello
Hello, world!
 t@cookbook:~/puppet/manifests$
```

Note that creating the file in `/tmp` did not require special permissions. We did not run Puppet via `sudo`. Puppet need not be run through `sudo`; there are cases where running via an unprivileged user can be useful.

There's more...

When several people are working on a code base, it's easy for style inconsistencies to creep in. Fortunately, there's a tool available which can automatically check your code for compliance with the style guide: `puppet-lint`. We'll see how to use this in the next section.

Checking your manifests with Puppet-lint

The puppetlabs official style guide outlines a number of style conventions for Puppet code, some of which we've touched on in the preceding section. For example, according to the style guide, manifests:

- Must use two-space soft tabs
- Must not use literal tab characters
- Must not contain trailing white space
- Should not exceed an 80 character line width
- Should align parameter arrows (`=>`) within blocks

Following the style guide will make sure that your Puppet code is easy to read and maintain, and if you're planning to release your code to the public, style compliance is essential.

The `puppet-lint` tool will automatically check your code against the style guide. The next section explains how to use it.

Getting ready

Here's what you need to do to install Puppet-lint:

1. We'll install Puppet-lint using the gem provider because the gem version is much more up to date than the APT or RPM packages available. Create a `puppet-lint.pp` manifest as shown in the following code snippet:

```
package {'puppet-lint':
  ensure => 'installed',
  provider => 'gem',
}
```

2. Run `puppet apply` on the `puppet-lint.pp` manifest, as shown in the following command:

```
t@cookbook ~$ puppet apply puppet-lint.pp Notice: Compiled catalog
for node1.example.com in environment production in 0.42 seconds
Notice: /Stage[main]/Main/Package[puppet-lint]/ensure: created
Notice: Finished catalog run in 2.96 seconds
t@cookbook ~$ gem list puppet-lint *** LOCAL GEMS *** puppet-lint
(1.0.1)
```

How to do it...

Follow these steps to use Puppet-lint:

1. Choose a Puppet manifest file that you want to check with Puppet-lint, and run the following command:

```
t@cookbook ~$ puppet-lint puppet-lint.pp
WARNING: indentation of => is not properly aligned on line 2
ERROR: trailing whitespace found on line 4
```

2. As you can see, Puppet-lint found a number of problems with the manifest file. Correct the errors, save the file, and rerun Puppet-lint to check that all is well. If successful, you'll see no output:

```
t@cookbook ~$ puppet-lint puppet-lint.pp
t@cookbook ~$
```

There's more...

You can find out more about Puppet-lint at `https://github.com/rodjek/puppet-lint`.

Should you follow Puppet style guide and, by extension, keep your code lint-clean? It's up to you, but here are a couple of things to think about:

- It makes sense to use some style conventions, especially when you're working collaboratively on code. Unless you and your colleagues can agree on standards for whitespace, tabs, quoting, alignment, and so on, your code will be messy and difficult to read or maintain.
- If you're choosing a set of style conventions to follow, the logical choice would be that issued by puppetlabs and adopted by the community for use in public modules.

Having said that, it's possible to tell Puppet-lint to ignore certain checks if you've chosen not to adopt them in your codebase. For example, if you don't want Puppet-lint to warn you about code lines exceeding 80 characters, you can run Puppet-lint with the following option:

```
t@cookbook ~$ puppet-lint --no-80chars-check
```

Run `puppet-lint --help` to see the complete list of check configuration commands.

See also

- The *Automatic syntax checking with Git hooks* recipe in *Chapter 2*, *Puppet Infrastructure*
- The *Testing your Puppet manifests with rspec-puppet* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*

Using modules

One of the most important things you can do to make your Puppet manifests clearer and more maintainable is to organize them into modules.

Modules are self-contained bundles of Puppet code that include all the files necessary to implement a thing. Modules may contain flat files, templates, Puppet manifests, custom fact declarations, augeas lenses, and custom Puppet types and providers.

Separating things into modules makes it easier to reuse and share code; it's also the most logical way to organize your manifests. In this example, we'll create a module to manage memcached, a memory caching system commonly used with web applications.

How to do it...

Following are the steps to create an example module:

1. We will use Puppet's module subcommand to create the directory structure for our new module:

```
t@cookbook:~$ mkdir -p .puppet/modules
t@cookbook:~$ cd .puppet/modules
t@cookbook:~/.puppet/modules$ puppet module generate thomas-
memcached
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]
--> Who wrote this module?  [thomas]
--> What license does this module code fall under?  [Apache 2.0]
--> How would you describe this module in a single sentence?
--> A module to install memcached Where is this module's source
code repository?
--> Where can others go to learn more about this module?
--> Where can others go to file issues about this module?
-->
----------------------------------------
{
  "name": "thomas-memcached",
  "version": "0.1.0",
  "author": "thomas",
  "summary": "A module to install memcached",
  "license": "Apache 2.0",
  "source": "",
  "issues_url": null,
  "project_page": null,
  "dependencies": [
    {
      "version_range": ">= 1.0.0",
      "name": "puppetlabs-stdlib"
    }
  ]
}
```

```
---------------------------------------
About to generate this metadata; continue? [n/Y]
--> y
Notice: Generating module at /home/thomas/.puppet/modules/thomas-
memcached...
Notice: Populating ERB templates...
Finished; module generated in thomas-memcached.
thomas-memcached/manifests
thomas-memcached/manifests/init.pp
thomas-memcached/spec
thomas-memcached/spec/classes
thomas-memcached/spec/classes/init_spec.rb
thomas-memcached/spec/spec_helper.rb
thomas-memcached/README.md
thomas-memcached/metadata.json
thomas-memcached/Rakefile
thomas-memcached/tests
thomas-memcached/tests/init.pp
```

This command creates the module directory and creates some empty files as starting points. To use the module, we'll create a symlink to the module name (memcached).

```
t@cookbook:~/.puppet/modules$ ln –s thomas-memcached memcached
```

2. Now, edit `memcached/manifests/init.pp` and change the class definition at the end of the file to the following. Note that `puppet module generate` created many lines of comments; in a production module you would want to edit those default comments:

```
class memcached {
  package { 'memcached':
    ensure => installed,
  }

  file { '/etc/memcached.conf':
    source  => 'puppet:///modules/memcached/memcached.conf',
    owner   => 'root',
    group   => 'root',
    mode    => '0644',
    require => Package['memcached'],
  }
```

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

3. Create the `modules/thomas-memcached/files` directory and then create a file named `memcached.conf` with the following contents:

```
-m 64
-p 11211
-u nobody
-l 127.0.0.1
```

4. Change your `site.pp` file to the following:

```
node default {
  include memcached
}
```

5. We would like this module to install memcached. We'll need to run Puppet with root privileges, and we'll use sudo for that. We'll need Puppet to be able to find the module in our home directory; we can specify this on the command line when we run Puppet as shown in the following code snippet:

```
t@cookbook:~$ sudo puppet apply --modulepath=/home/thomas/.puppet/
modules /home/thomas/.puppet/manifests/site.pp
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.33 seconds
Notice: /Stage[main]/Memcached/File[/etc/memcached.conf]/content:
content changed '{md5}a977521922a151c959ac953712840803' to '{md5}9
429eff3e3354c0be232a020bcf78f75'
Notice: Finished catalog run in 0.11 seconds
```

6. Check whether the new service is running:

```
t@cookbook:~$ sudo service memcached status
[ ok ] memcached is running.
```

How it works...

When we created the module using Puppet's module generate command, we used the name `thomas-memcached`. The name before the hyphen is your username or your username on Puppet forge (an online repository of modules). Since we want Puppet to be able to find the module by the name `memcached`, we make a symbolic link between `thomas-memcached` and `memcached`.

Modules have a specific directory structure. Not all of these directories need to be present, but if they are, this is how they should be organized:

```
modules/
  └MODULE_NAME/               never use a dash (-) in a module name
     └examples/               example usage of the module
     └files/                  flat files used by the module
     └lib/
        └facter/              define new facts for facter
        └puppet/
           └parser/
              └functions/     define a new puppet function, like
sort()
           └provider/    define a provider for a new or existing type
           └util/        define helper functions (in ruby)
           └type/        define a new type in puppet
     └manifests/
        └init.pp          class MODULE_NAME { }
     └spec/ rSpec         tests
     └templates/          erb template files used by the module
```

All manifest files (those containing Puppet code) live in the manifests directory. In our example, the `memcached` class is defined in the `manifests/init.pp` file, which will be imported automatically.

Inside the `memcached` class, we refer to the `memcached.conf` file:

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

The preceding `source` parameter tells Puppet to look for the file in:

```
MODULEPATH/   (/home/thomas/.puppet/modules)
  └memcached/
    └files/
      └memcached.conf
```

There's more...

Learn to love modules because they'll make your Puppet life a lot easier. They're not complicated, however, practice and experience will help you judge when things should be grouped into modules, and how best to arrange your module structure. Modules can hold more than manifests and files as we'll see in the next two sections.

Templates

If you need to use a template as a part of the module, place it in the module's templates directory and refer to it as follows:

```
file { '/etc/memcached.conf':
  content => template('memcached/memcached.conf.erb'),
}
```

Puppet will look for the file in:

```
MODULEPATH/memcached/templates/memcached.conf.erb
```

Facts, functions, types, and providers

Modules can also contain custom facts, custom functions, custom types, and providers.

For more information about these, refer to *Chapter 9*, *External Tools and the Puppet Ecosystem*.

Third-party modules

You can download modules provided by other people and use them in your own manifests just like the modules you create. For more on this, see Using Public Modules recipe in *Chapter 7*, *Managing Applications*.

Module organization

For more details on how to organize your modules, see puppetlabs website:

`http://docs.puppetlabs.com/puppet/3/reference/modules_fundamentals.html`

See also

- The *Creating custom facts* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*
- The *Using public modules* recipe in *Chapter 7*, *Managing Applications*
- The *Creating your own resource types* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*
- The *Creating your own providers* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*

Using standard naming conventions

Choosing appropriate and informative names for your modules and classes will be a big help when it comes to maintaining your code. This is even truer if other people need to read and work on your manifests.

How to do it...

Here are some tips on how to name things in your manifests:

1. Name modules after the software or service they manage, for example, `apache` or `haproxy`.
2. Name classes within modules (subclasses) after the function or service they provide to the module, for example, `apache::vhosts` or `rails::dependencies`.
3. If a class within a module disables the service provided by that module, name it `disabled`. For example, a class that disables Apache should be named `apache::disabled`.
4. Create a roles and profiles hierarchy of modules. Each node should have a single role consisting of one or more profiles. Each profile module should configure a single service.
5. The module that manages users should be named `user`.
6. Within the user module, declare your virtual users within the class `user::virtual` (for more on virtual users and other resources, see the *Using virtual resources* recipe in *Chapter 5*, *Users and Virtual Resources*).
7. Within the user module, subclasses for particular groups of users should be named after the group, for example, `user::sysadmins` or `user::contractors`.
8. When using Puppet to deploy the config files for different services, name the file after the service, but with a suffix indicating what kind of file it is, for example:
   - Apache init script: `apache.init`
   - Logrotate config snippet for Rails: `rails.logrotate`
   - Nginx vhost file for mywizzoapp: `mywizzoapp.vhost.nginx`
   - MySQL config for standalone server: `standalone.mysql`
9. If you need to deploy a different version of a file depending on the operating system release, for example, you can use a naming convention like the following:

   ```
   memcached.lucid.conf
   memcached.precise.conf
   ```

10. You can have Puppet automatically select the appropriate version as follows:

    ```
    source = > "puppet:///modules/memcached
      /memcached.${::lsbdistrelease}.conf",
    ```

11. If you need to manage, for example, different Ruby versions, name the class after the version it is responsible for, for example, `ruby192` or `ruby186`.

There's more...

Puppet community maintains a set of best practice guidelines for your Puppet infrastructure, which includes some hints on naming conventions:

`http://docs.puppetlabs.com/guides/best_practices.html`

Some people prefer to include multiple classes on a node by using a comma-separated list, rather than separate `include` statements, for example:

```
node 'server014' inherits 'server' {
  include mail::server, repo::gem, repo::apt, zabbix
}
```

This is a matter of style, but I prefer to use separate `include` statements, one on a line, because it makes it easier to copy and move around class inclusions between nodes without having to tidy up the commas and indentation every time.

I mentioned inheritance in a couple of the preceding examples; if you're not sure what this is, don't worry, I'll explain this in detail in the next chapter.

Using inline templates

Templates are a powerful way of using **Embedded Ruby** (**ERB**) to help build config files dynamically. You can also use ERB syntax directly without having to use a separate file by calling the `inline_template` function. ERB allows you to use conditional logic, iterate over arrays, and include variables.

How to do it...

Here's an example of how to use `inline_template`:

Pass your Ruby code to `inline_template` within Puppet manifest, as follows:

```
cron { 'chkrootkit':
  command => '/usr/sbin/chkrootkit >
    /var/log/chkrootkit.log 2>&1',
  hour    => inline_template('<%= @hostname.sum % 24 %>'),
  minute  => '00',
}
```

How it works...

Anything inside the string passed to `inline_template` is executed as if it were an ERB template. That is, anything inside the `<%=` and `%>` delimiters will be executed as Ruby code, and the rest will be treated as a string.

In this example, we use `inline_template` to compute a different hour for this cron resource (a scheduled job) for each machine, so that the same job does not run at the same time on all machines. For more on this technique, see the *Distributing cron jobs efficiently* recipe in *Chapter 6*, *Managing Resources and Files*.

There's more...

In ERB code, whether inside a template file or an `inline_template` string, you can access your Puppet variables directly by name using an `@` prefix, if they are in the current scope or the top scope (facts):

```
<%= @fqdn %>
```

To reference variables in another scope, use `scope.lookupvar`, as follows:

```
<%= "The value of something from otherclass is " +
  scope.lookupvar('otherclass::something') %>
```

You should use inline templates sparingly. If you really need to use some complicated logic in your manifest, consider using a custom function instead (see the *Creating custom functions* recipe in *Chapter 9*, *External Tools and the Puppet Ecosystem*).

See also

- The *Using ERB templates* recipe in *Chapter 4*, *Working with Files and Packages*
- The *Using array iteration in templates* recipe in *Chapter 4*, *Working with Files and Packages*

Iterating over multiple items

Arrays are a powerful feature in Puppet; wherever you want to perform the same operation on a list of things, an array may be able to help. You can create an array just by putting its content in square brackets:

```
$lunch = [ 'franks', 'beans', 'mustard' ]
```

How to do it...

Here's a common example of how arrays are used:

1. Add the following code to your manifest:

```
$packages = [ 'ruby1.8-dev',
  'ruby1.8',
  'ri1.8',
  'rdoc1.8',
  'irb1.8',
  'libreadline-ruby1.8',
  'libruby1.8',
  'libopenssl-ruby' ]

package { $packages: ensure => installed }
```

2. Run Puppet and note that each package should now be installed.

How it works...

Where Puppet encounters an array as the name of a resource, it creates a resource for each element in the array. In the example, a new package resource is created for each of the packages in the `$packages` array, with the same parameters (`ensure => installed`). This is a very compact way to instantiate many similar resources.

There's more...

Although arrays will take you a long way with Puppet, it's also useful to know about an even more flexible data structure: the hash.

Using hashes

A hash is like an array, but each of the elements can be stored and looked up by name (referred to as the key), for example (`hash.pp`):

```
$interface = {
  'name' => 'eth0',
  'ip'   => '192.168.0.1',
  'mac'  => '52:54:00:4a:60:07'
}
notify { "(${interface['ip']}) at ${interface['mac']} on
  ${interface['name']}": }
```

When we run Puppet on this, we see the following notify in the output:

```
t@cookbook:~/.puppet/manifests$ puppet apply hash.pp
Notice: (192.168.0.1) at 52:54:00:4a:60:07 on etho
```

Hash values can be anything that you can assign to variables, strings, function calls, expressions, and even other hashes or arrays. Hashes are useful to store a bunch of information about a particular thing because by accessing each element of the hash using a key, we can quickly find the information for which we are looking.

Creating arrays with the split function

You can declare literal arrays using square brackets, as follows:

```
define lunchprint() {
  notify { "Lunch included ${name}":}": }
}

$lunch = ['egg', 'beans', 'chips']
lunchprint { $lunch: }
```

Now, when we run Puppet on the preceding code, we see the following notice messages in the output:

```
t@mylaptop ~ $ puppet apply lunchprint.pp
...
Notice: Lunch included chips
Notice: Lunch included beans
Notice: Lunch included egg
```

However, Puppet can also create arrays for you from strings, using the `split` function, as follows:

```
$menu = 'egg beans chips'
$items = split($menu, ' ')
lunchprint { $items: }
```

Running `puppet apply` against this new manifest, we see the same messages in the output:

```
t@mylaptop ~ $ puppet apply lunchprint2.pp
...
Notice: Lunch included chips
Notice: Lunch included beans
Notice: Lunch included egg.
```

Note that `split` takes two arguments: the first argument is the string to be split. The second argument is the character to split on; in this example, a single space. As Puppet works its way through the string, when it encounters a space, it will interpret it as the end of one item and the beginning of the next. So, given the string `'egg beans chips'`, this will be split into three items.

The character to split on can be any character or string:

```
$menu = 'egg and beans and chips'
$items = split($menu, ' and ')
```

The character can also be a regular expression, for example, a set of alternatives separated by a | (pipe) character:

```
$lunch = 'egg:beans,chips'
$items = split($lunch, ':|,')
```

Writing powerful conditional statements

Puppet's `if` statement allows you to change the manifest behavior based on the value of a variable or an expression. With it, you can apply different resources or parameter values depending on certain facts about the node, for example, the operating system, or the memory size.

You can also set variables within the manifest, which can change the behavior of included classes. For example, nodes in data center A might need to use different DNS servers than nodes in data center B, or you might need to include one set of classes for an Ubuntu system, and a different set for other systems.

How to do it...

Here's an example of a useful conditional statement. Add the following code to your manifest:

```
if $::timezone == 'UTC' {
  notify { 'Universal Time Coordinated':}
} else {
  notify { "$::timezone is not UTC": }
}
```

How it works...

Puppet treats whatever follows an `if` keyword as an expression and evaluates it. If the expression evaluates to true, Puppet will execute the code within the curly braces.

Optionally, you can add an else branch, which will be executed if the expression evaluates to false.

There's more...

Here are some more tips on using `if` statements.

Elsif branches

You can add further tests using the `elsif` keyword, as follows:

```
if $::timezone == 'UTC' {
  notify { 'Universal Time Coordinated': }
} elsif $::timezone == 'GMT' {
  notify { 'Greenwich Mean Time': }
} else {
  notify { "$::timezone is not UTC": }
}
```

Comparisons

You can check whether two values are equal using the `==` syntax, as in our example:

```
if $::timezone == 'UTC' {

}
```

Alternatively, you can check whether they are not equal using `!=`:

```
if $::timezone != 'UTC' {
  …
}
```

You can also compare numeric values using `<` and `>`:

```
if $::uptime_days > 365 {
  notify { 'Time to upgrade your kernel!': }
}
```

To test whether a value is greater (or less) than or equal to another value, use `<=` or `>=`:

```
if $::mtu_eth0 <= 1500 {
  notify {"Not Jumbo Frames": }
}
```

Combining expressions

You can put together the kind of simple expressions described previously into more complex logical expressions, using `and`, `or`, and `not`:

```
if ($::uptime_days > 365) and ($::kernel == 'Linux') {
  …
}

if ($role == 'webserver') and ( ($datacenter == 'A') or ($datacenter
== 'B') ) {
  …
}
```

See also

- The *Using the in operator* recipe in this chapter
- The *Using selectors and case statements* recipe in this chapter

Using regular expressions in if statements

Another kind of expression you can test in `if` statements and other conditionals is the regular expression. A regular expression is a powerful way to compare strings using pattern matching.

How to do it...

This is one example of using a regular expression in a conditional statement. Add the following to your manifest:

```
if $::architecture =~ /64/ {
  notify { '64Bit OS Installed': }
} else {
  notify { 'Upgrade to 64Bit': }
  fail('Not 64 Bit')
}
```

How it works...

Puppet treats the text supplied between the forward slashes as a regular expression, specifying the text to be matched. If the match succeeds, the `if` expression will be true and so the code between the first set of curly braces will be executed. In this example, we used a regular expression because different distributions have different ideas on what to call `64bit`; some use `amd64`, while others use `x86_64`. The only thing we can count on is the presence of the number 64 within the fact. Some facts that have version numbers in them are treated as strings to Puppet. For instance, `$::facterversion`. On my test system, this is `2.0.1`, but when I try to compare that with `2`, Puppet fails to make the comparison:

```
Error: comparison of String with 2 failed at /home/thomas/.puppet/
manifests/version.pp:1 on node cookbook.example.com
```

If you wanted instead to do something if the text does not match, use `!~` rather than `=~`:

```
if $::kernel !~ /Linux/ {
  notify { 'Not Linux, could be Windows, MacOS X, AIX, or ?': }
}
```

There's more...

Regular expressions are very powerful, but can be difficult to understand and debug. If you find yourself using a regular expression so complex that you can't see at a glance what it does, think about simplifying your design to make it easier. However, one particularly useful feature of regular expressions is the ability to capture patterns.

Capturing patterns

You can not only match text using a regular expression, but also capture the matched text and store it in a variable:

```
$input = 'Puppet is better than manual configuration'
if $input =~ /(.*) is better than (.*)/ {
  notify { "You said '${0}'. Looks like you're comparing ${1}
    to ${2}!": }
}
```

The preceding code produces this output:

```
You said 'Puppet is better than manual configuration'. Looks like you're comparing Puppet
to manual configuration!
```

The variable `$0` stores the whole matched text (assuming the overall match succeeded). If you put brackets around any part of the regular expression, it creates a group, and any matched groups will also be stored in variables. The first matched group will be `$1`, the second `$2`, and so on, as shown in the preceding example.

Regular expression syntax

Puppet's regular expression syntax is the same as Ruby's, so resources that explain Ruby's regular expression syntax will also help you with Puppet. You can find a good introduction to Ruby's regular expression syntax at this website:

`http://www.tutorialspoint.com/ruby/ruby_regular_expressions.htm`.

See also

- Refer to the *Using regular expression substitutions* recipe in this chapter

Using selectors and case statements

Although you could write any conditional statement using `if`, Puppet provides a couple of extra forms to help you express conditionals more easily: the selector and the `case` statement.

How to do it...

Here are some examples of selector and `case` statements:

1. Add the following code to your manifest:

```
$systemtype = $::operatingsystem ? {
  'Ubuntu' => 'debianlike',
  'Debian' => 'debianlike',
  'RedHat' => 'redhatlike',
  'Fedora' => 'redhatlike',
  'CentOS' => 'redhatlike',
  default  => 'unknown',
}

notify { "You have a ${systemtype} system": }
```

2. Add the following code to your manifest:

```
class debianlike {
  notify { 'Special manifest for Debian-like systems': }
}

class redhatlike {
  notify { 'Special manifest for RedHat-like systems': }
}

case $::operatingsystem {
  'Ubuntu',
```

```
    'Debian': {
      include debianlike
    }
    'RedHat',
    'Fedora',
    'CentOS',
    'Springdale': {
      include redhatlike
    }
    default: {
      notify { "I don't know what kind of system you have!":
      }
    }
  }
```

How it works...

Our example demonstrates both the selector and the `case` statement, so let's see in detail how each of them works.

Selector

In the first example, we used a selector (the `?` operator) to choose a value for the `$systemtype` variable depending on the value of `$::operatingsystem`. This is similar to the ternary operator in C or Ruby, but instead of choosing between two possible values, you can have as many values as you like.

Puppet will compare the value of `$::operatingsystem` to each of the possible values we have supplied in Ubuntu, Debian, and so on. These values could be regular expressions (for example, for a partial string match, or to use wildcards), but in our case, we have just used literal strings.

As soon as it finds a match, the selector expression returns whatever value is associated with the matching string. If the value of `$::operatingsystem` is Fedora, for example, the selector expression will return the `redhatlike` string and this will be assigned to the variable `$systemtype`.

Case statement

Unlike selectors, the `case` statement does not return a value. `case` statements come in handy when you want to execute different code depending on the value of some expression. In our second example, we used the `case` statement to include either the `debianlike` or `redhatlike` class, depending on the value of `$::operatingsystem`.

Again, Puppet compares the value of `$::operatingsystem` to a list of potential matches. These could be regular expressions or strings, or as in our example, comma-separated lists of strings. When it finds a match, the associated code between curly braces is executed. So, if the value of `$::operatingsystem` is `Ubuntu`, then the code including `debianlike` will be executed.

There's more...

Once you've got a grip of the basic use of selectors and `case` statements, you may find the following tips useful.

Regular expressions

As with `if` statements, you can use regular expressions with selectors and `case` statements, and you can also capture the values of the matched groups and refer to them using `$1`, `$2`, and so on:

```
case $::lsbdistdescription {
  /Ubuntu (.+)/: {
    notify { "You have Ubuntu version ${1}": }
  }
  /CentOS (.+)/: {
    notify { "You have CentOS version ${1}": }
  }
  default: {}
}
```

Defaults

Both selectors and `case` statements let you specify a default value, which is chosen if none of the other options match (the style guide suggests you always have a default clause defined):

```
$lunch = 'Filet mignon.'
$lunchtype =  $lunch ? {
  /fries/ => 'unhealthy',
  /salad/ => 'healthy',
  default => 'unknown',
}

notify { "Your lunch was ${lunchtype}": }
```

The output is as follows:

```
t@mylaptop ~ $ puppet apply lunchtype.pp
Notice: Your lunch was unknown
```

```
Notice: /Stage[main]/Main/Notify[Your lunch was unknown]/message: defined
'message' as 'Your lunch was unknown'
```

When the default action shouldn't normally occur, use the `fail()` function to halt the Puppet run.

Using the in operator

The `in` operator tests whether one string contains another string. Here's an example:

```
if 'spring' in 'springfield'
```

The preceding expression is true if the `spring` string is a substring of `springfield`, which it is. The `in` operator can also test for membership of arrays as follows:

```
if $crewmember in ['Frank', 'Dave', 'HAL' ]
```

When `in` is used with a hash, it tests whether the string is a key of the hash:

```
$ifaces = { 'lo'   => '127.0.0.1',
            'eth0' => '192.168.0.1' }
if 'eth0' in $ifaces {
  notify { "eth0 has address ${ifaces['eth0']}": }
}
```

How to do it...

The following steps will show you how to use the `in` operator:

1. Add the following code to your manifest:

```
if $::operatingsystem in [ 'Ubuntu', 'Debian' ] {
  notify { 'Debian-type operating system detected': }
} elsif $::operatingsystem in [ 'RedHat', 'Fedora', 'SuSE',
  'CentOS' ] {
  notify { 'RedHat-type operating system detected': }
} else {
  notify { 'Some other operating system detected': }
}
```

2. Run Puppet:

```
t@cookbook:~/.puppet/manifests$ puppet apply in.pp
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.03 seconds
Notice: Debian-type operating system detected
```

```
Notice: /Stage[main]/Main/Notify[Debian-type operating system
detected]/message: defined 'message' as 'Debian-type operating
system detected'
Notice: Finished catalog run in 0.02 seconds
```

There's more...

The value of an `in` expression is Boolean (true or false) so you can assign it to a variable:

```
$debianlike = $::operatingsystem in [ 'Debian', 'Ubuntu' ]

if $debianlike {
  notify { 'You are in a maze of twisty little packages, all alike': }
}
```

Using regular expression substitutions

Puppet's `regsubst` function provides an easy way to manipulate text, search and replace expressions within strings, or extract patterns from strings. We often need to do this with data obtained from a fact, for example, or from external programs.

In this example, we'll see how to use `regsubst` to extract the first three octets of an IPv4 address (the network part, assuming it's a `/24` class C address).

How to do it...

Follow these steps to build the example:

1. Add the following code to your manifest:

```
$class_c = regsubst($::ipaddress, '(.*)\..*', '\1.0')
notify { "The network part of ${::ipaddress} is ${class_c}": }
```

2. Run Puppet:

```
t@cookbook:~/.puppet/manifests$ puppet apply ipaddress.pp
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.02 seconds
Notice: The network part of 192.168.122.148 is
  192.168.122.0
Notice: /Stage[main]/Main/Notify[The network part of
192.168.122.148 is
  192.168.122.0]/message: defined 'message' as 'The network part
of 192.168.122.148 is
```

```
  192.168.122.0'
Notice: Finished catalog run in 0.03 seconds
```

How it works...

The `regsubst` function takes at least three parameters: source, pattern, and replacement. In our example, we specified the source string as `$::ipaddress`, which, on this machine, is as follows:

```
192.168.122.148
```

We specify the `pattern` function as follows:

```
(.*)\..*
```

We specify the `replacement` function as follows:

```
\1.0
```

The pattern captures all of the string up to the last period (`\.`) in the `\1` variable. We then match on `.*`, which matches everything to the end of the string, so when we replace the string at the end with `\1.0`, we end up with only the network portion of the IP address, which evaluates to the following:

```
192.168.122.0
```

We could have got the same result in other ways, of course, including the following:

```
$class_c = regsubst($::ipaddress, '\.\d+$', '.0')
```

Here, we only match the last octet and replace it with `.0`, which achieves the same result without capturing.

There's more...

The `pattern` function can be any regular expression, using the same (Ruby) syntax as regular expressions in `if` statements.

See also

- The *Importing dynamic information* recipe in *Chapter 3*, *Writing Better Manifests*
- The *Getting information about the environment* recipe in *Chapter 3*, *Writing Better Manifests*
- The *Using regular expressions in if statements* recipe in this chapter

Using the future parser

Puppet language is evolving at the moment; many features that are expected to be included in the next major release (4) are available if you enable the future parser.

Getting ready

- Ensure that the `rgen` gem is installed.
- Set `parser = future` in the `[main]` section of your `puppet.conf`(`/etc/puppet/puppet.conf` for open source Puppet as `root`, `/etc/puppetlabs/puppet/puppet.conf` for Puppet `Enterprise`, and `~/.puppet/puppet.conf` for a non-root user running puppet).
- To temporarily test with the future parser, use `--parser=future` on the command line.

How to do it...

Many of the experimental features deal with how code is evaluated, for example, in an earlier example we compared the value of the `$::facterversion` fact with a number, but the value is treated as a string so the code fails to compile. Using the future parser, the value is converted and no error is reported as shown in the following command line output:

```
t@cookbook:~/.puppet/manifests$ puppet apply --parser=future version.pp
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
```

Appending to and concatenating arrays

You can concatenate arrays with the `+` operator or append them with the `<<` operator. In the following example, we use the ternary operator to assign a specific package name to the `$apache` variable. We then append that value to an array using the `<<` operator:

```
$apache = $::osfamily ? {
  'Debian' => 'apache2',
  'RedHat' => 'httpd'
}
$packages = ['memcached'] << $apache
package {$packages: ensure => installed}
```

If we have two arrays, we can use the `+` operator to concatenate the two arrays. In this example, we define an array of system administrators (`$sysadmins`) and another array of application owners (`$appowners`). We can then concatenate the array and use it as an argument to our allowed users:

```
$sysadmins = [ 'thomas','john','josko' ]
$appowners = [ 'mike', 'patty', 'erin' ]
$users = $sysadmins + $appowners
notice ($users)
```

When we apply this manifest, we see that the two arrays have been joined as shown in the following command line output:

```
t@cookbook:~/.puppet/manifests$ puppet apply --parser=future concat.pp
Notice: [thomas, john, josko, mike, patty, erin]
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
```

Merging Hashes

If we have two hashes, we can merge them using the same `+` operator we used for arrays. Consider our `$interfaces` hash from a previous example; we can add another interface to the hash:

```
$iface = {
  'name' => 'eth0',
  'ip'   => '192.168.0.1',
  'mac'  => '52:54:00:4a:60:07'
}  + {'route' => '192.168.0.254'}
notice ($iface)
```

When we apply this manifest, we see that the route attribute has been merged into the hash (your results may differ, the order in which the hash prints is unpredictable), as follows:

```
t@cookbook:~/.puppet/manifests$ puppet apply --parser=future hash2.pp
Notice: {route => 192.168.0.254, name => eth0, ip => 192.168.0.1, mac =>
52:54:00:4a:60:07}
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
```

Lambda functions

Lambda functions are iterators applied to arrays or hashes. You iterate through the array or hash and apply an iterator function such as `each`, `map`, `filter`, `reduce`, or `slice` to each element of the array or key of the hash. Some of the lambda functions return a calculated array or value; others such as `each` only return the input array or hash.

Lambda functions such as `map` and `reduce` use temporary variables that are thrown away after the lambda has finished. Use of lambda functions is something best shown by example. In the next few sections, we will show an example usage of each of the lambda functions.

Reduce

Reduce is used to reduce the array to a single value. This can be used to calculate the maximum or minimum of the array, or in this case, the sum of the elements of the array:

```
$count = [1,2,3,4,5]
$sum = reduce($count) | $total, $i | { $total + $i }
notice("Sum is $sum")
```

This preceding code will compute the sum of the `$count` array and store it in the `$sum` variable, as follows:

```
t@cookbook:~/.puppet/manifests$ puppet apply --parser future lambda.pp
Notice: Sum is 15
Notice: Compiled catalog for cookbook.example.com in environment
production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
```

Filter

Filter is used to filter the array or hash based upon a test within the lambda function. For instance to filter our `$count` array as follows:

```
$filter = filter ($count) | $i | { $i > 3 }
notice("Filtered array is $filter")
```

When we apply this manifest, we see that only elements 4 and 5 are in the result:

```
Notice: Filtered array is [4, 5]
```

Map

Map is used to apply a function to each element of the array. For instance, if we wanted (for some unknown reason) to compute the square of all the elements of the array, we would use `map` as follows:

```
$map = map ($count) | $i | { $i * $i }
notice("Square of array is $map")
```

The result of applying this manifest is a new array with every element of the original array squared (multiplied by itself), as shown in the following command line output:

```
Notice: Square of array is [1, 4, 9, 16, 25]
```

Slice

Slice is useful when you have related values stored in the same array in a sequential order. For instance, if we had the destination and port information for a firewall in an array, we could split them up into pairs and perform operations on those pairs:

```
$firewall_rules = ['192.168.0.1','80','192.168.0.10','443']
slice ($firewall_rules,2) |$ip, $port| { notice("Allow $ip on
  $port") }
```

When applied, this manifest will produce the following notices:

```
Notice: Allow 192.168.0.1 on 80
Notice: Allow 192.168.0.10 on 443
```

To make this a useful example, create a new firewall resource within the block of the slice instead of notice:

```
slice ($firewall_rules,2) |$ip, $port| {
  firewall {"$port from $ip":
    dport  => $port,
    source => "$ip",
    action => 'accept',
  }
}
```

Each

Each is used to iterate over the elements of the array but lacks the ability to capture the results like the other functions. Each is the simplest case where you simply wish to do something with each element of the array, as shown in the following code snippet:

```
each ($count) |$c| { notice($c) }
```

As expected, this executes the `notice` for each element of the `$count` array, as follows:

```
Notice: 1
Notice: 2
Notice: 3
Notice: 4
Notice: 5
```

Other features

There are other new features of Puppet language available when using the future parser. Some increase readability or compactness of code. For more information, refer to the documentation on puppetlabs website at `http://docs.puppetlabs.com/puppet/latest/reference/experiments_future.html`.

2
Puppet Infrastructure

"Computers in the future may have as few as 1,000 vacuum tubes and weigh only 1.5 tons."

— Popular Mechanics, 1949

In this chapter, we will cover:

- Installing Puppet
- Managing your manifests with Git
- Creating a decentralized Puppet architecture
- Writing a papply script
- Running Puppet from cron
- Bootstrapping Puppet with bash
- Creating a centralized Puppet infrastructure
- Creating certificates with multiple DNS names
- Running Puppet from passenger
- Setting up the environment
- Configuring PuppetDB
- Configuring Hiera
- Setting-node specific data with Hiera
- Storing secret data with hiera-gpg
- Using MessagePack serialization
- Automatic syntax checking with Git hooks
- Pushing code around with Git
- Managing environments with Git

Introduction

In this chapter, we will cover how to deploy Puppet in a centralized and decentralized manner. With each approach, we'll see a combination of best practices, my personal experience, and community solutions.

We'll configure and use both PuppetDB and Hiera. PuppetDB is used with exported resources, which we will cover in *Chapter 5*, *Users and Virtual Resources*. Hiera is used to separate variable data from Puppet code.

Finally, I'll introduce Git and see how to use Git to organize our code and our infrastructure.

Because Linux distributions, such as Ubuntu, Red Hat, and CentOS, differ in the specific details of package names, configuration file paths, and many other things, I have decided that for reasons of space and clarity the best approach for this book is to pick one distribution (*Debian 7* named as *Wheezy*) and stick to that. However, Puppet runs on most popular operating systems, so you should have very little trouble adapting the recipes to your own favorite OS and distribution.

At the time of writing, Puppet 3.7.2 is the latest stable version available, this is the version of Puppet used in the book. The syntax of Puppet commands changes often, so be aware that while older versions of Puppet are still perfectly usable, they may not support all of the features and syntax described in this book. As we saw in *Chapter 1*, *Puppet Language and Style*, the future parser showcases features of the language scheduled to become default in Version 4 of Puppet.

Installing Puppet

In *Chapter 1*, *Puppet Language and Style*, we installed Puppet as a rubygem using the gem install. When deploying to several nodes, this may not be the best approach. Using the package manager of your chosen distribution is the best way to keep your Puppet versions similar on all of the nodes in your deployment. Puppet labs maintain repositories for APT-based and YUM-based distributions.

Getting ready

If your Linux distribution uses APT for package management, go to `http://apt.puppetlabs.com/` and download the appropriate Puppet labs release package for your distribution. For our wheezy cookbook node, we will use `http://apt.puppetlabs.com/puppetlabs-release-wheezy.deb`.

If you are using a Linux distribution that uses YUM for package management, go to `http://yum.puppetlabs.com/` and download the appropriate Puppet labs release package for your distribution.

How to do it...

1. Once you have found the appropriate Puppet labs release package for your distribution, the steps to install Puppet are the same for either APT or YUM:
    - Install Puppet labs release package
    - Install Puppet package
2. Once you have installed Puppet, verify the version of Puppet as shown in the following example:

```
t@ckbk:~ puppet --version 3.7.2
```

Now that we have a method to install Puppet on our nodes, we need to turn our attention to keeping our Puppet manifests organized. In the next section, we will see how to use Git to keep our code organized and consistent.

Managing your manifests with Git

It's a great idea to put your Puppet manifests in a version control system such as Git or Subversion (Git is the de facto standard for Puppet). This gives you several advantages:

- You can undo changes and revert to any previous version of your manifest
- You can experiment with new features using a branch
- If several people need to make changes to the manifests, they can make them independently, in their own working copies, and then merge their changes later
- You can use the `git log` feature to see what was changed, and when (and by whom)

Getting ready

In this section, we'll import your existing manifest files into Git. If you have created a Puppet directory in a previous section use that, otherwise, use your existing manifest directory.

In this example, we'll create a new Git repository on a server accessible from all our nodes. There are several steps we need to take to have our code held in a Git repository:

1. Install Git on a central server.
2. Create a user to run Git and own the repository.
3. Create a repository to hold the code.
4. Create **SSH** keys to allow key-based access to the repository.
5. Install Git on a node and download the latest version from our Git repository.

How to do it...

Follow these steps:

1. First, install Git on your Git server (`git.example.com` in our example). The easiest way to do this is using Puppet. Create the following manifest, call it `git.pp`:

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

2. Apply this manifest using `puppet apply git.pp`, this will install Git.
3. Next, create a Git user that the nodes will use to log in and retrieve the latest code. Again, we'll do this with puppet. We'll also create a directory to hold our repository (`/home/git/repos`) as shown in the following code snippet:

```
group { 'git':
  gid => 1111,
}
user {'git':
  uid => 1111,
  gid => 1111,
  comment => 'Git User',
  home => '/home/git',
  require => Group['git'],
}
file {'/home/git':
  ensure => 'directory',
  owner => 1111,
  group => 1111,
  require => User['git'],
}
file {'/home/git/repos':
  ensure => 'directory',
  owner => 1111,
  group => 1111,
  require => File['/home/git']
}
```

4. After applying that manifest, log in as the Git user and create an empty Git repository using the following command:

```
# sudo -iu git
git@git $ cd repos
git@git $ git init --bare puppet.git
Initialized empty Git repository in /home/git/repos/puppet.git/
```

5. Set a password for the Git user, we'll need to log in remotely after the next step:

```
[root@git ~]# passwd git
Changing password for user git.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

6. Now back on your local machine, create an `ssh` key for our nodes to use to update the repository:

```
t@mylaptop ~ $ cd .ssh
t@mylaptop ~/.ssh $ ssh-keygen -b 4096 -f git_rsa
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in git_rsa.
Your public key has been saved in git_rsa.pub.
The key fingerprint is:
87:35:0e:4e:d2:96:5f:e4:ce:64:4a:d5:76:c8:2b:e4 thomas@mylaptop
```

7. Now copy the newly created public key to the `authorized_keys` file. This will allow us to connect to the Git server using this new key:

```
t@mylaptop ~/.ssh $ ssh-copy-id -i git_rsa git@git.example.com
git@git.example.com's password:
Number of key(s) added: 1
```

8. Now try logging into the machine, with: "ssh 'git@git.example.com'" and check to make sure that only the key(s) you wanted were added.

9. Next, configure `ssh` to use your key when accessing the Git server and add the following to your `~/.ssh/config` file:

```
Host git git.example.com
  User git
  IdentityFile /home/thomas/.ssh/git_rsa
```

10. Clone the repo onto your machine into a directory named Puppet (substitute your server name if you didn't use `git.example.com`):

```
t@mylaptop ~$ git clone git@git.example.com:repos/puppet.git
Cloning into 'puppet'...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
```

We've created a Git repository; before we commit any changes to the repository, it's a good idea to set your name and e-mail in Git. Your name and e-mail will be appended to each commit you make.

11. When you are working in a large team, knowing who made a change is very important; for this, use the following code snippet:

```
t@mylaptop puppet$ git config --global user.email
"thomas@narrabilis.com"
t@mylaptop puppet$ git config --global user.name "Thomas
Uphill"
```

12. You can verify your Git settings using the following snippet:

```
t@mylaptop ~$ git config --global --list
user.name=Thomas Uphill
user.email=thomas@narrabilis.com
core.editor=vim
merge.tool=vimdiff
color.ui=true
push.default=simple
```

13. Now that we have Git configured properly, change directory to your repository directory and create a new site manifest as shown in the following snippet:

```
t@mylaptop ~$ cd puppet
t@mylaptop puppet$ mkdir manifests
t@mylaptop puppet$ vim manifests/site.pp
node default {
  include base
}
```

14. This site manifest will install our base class on every node; we will create the base class using the Puppet module as we did in *Chapter 1*, *Puppet Language and Style*:

```
t@mylaptop puppet$ mkdir modules
t@mylaptop puppet$ cd modules
t@mylaptop modules$ puppet module generate thomas-base
Notice: Generating module at /home/tuphill/puppet/modules/thomas-
base
thomas-base
thomas-base/Modulefile
thomas-base/README
thomas-base/manifests
thomas-base/manifests/init.pp
thomas-base/spec
thomas-base/spec/spec_helper.rb
```

```
thomas-base/tests
thomas-base/tests/init.pp
t@mylaptop modules$ ln -s thomas-base base
```

15. As a last step, we create a symbolic link between the `thomas-base` directory and `base`. Now to make sure our module does something useful, add the following to the body of the `base` class defined in `thomas-base/manifests/init.pp`:

```
class base {
  file {'/etc/motd':
    content => "${::fqdn}\nManaged by puppet ${::puppetversion}\n"
  }
}
```

16. Now add the new base module and site manifest to Git using `git add` and `git commit` as follows:

```
t@mylaptop modules$ cd ..
t@mylaptop puppet$ git add modules manifests
t@mylaptop puppet$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   manifests/site.pp
new file:   modules/base
new file:   modules/thomas-base/Modulefile
new file:   modules/thomas-base/README
new file:   modules/thomas-base/manifests/init.pp
new file:   modules/thomas-base/spec/spec_helper.rb
new file:   modules/thomas-base/tests/init.pp
t@mylaptop puppet$ git commit -m "Initial commit with simple base
module"
[master (root-commit) 3e1f837] Initial commit with simple base
module
 7 files changed, 102 insertions(+)
 create mode 100644 manifests/site.pp
 create mode 120000 modules/base
 create mode 100644 modules/thomas-base/Modulefile
 create mode 100644 modules/thomas-base/README
```

```
 create mode 100644 modules/thomas-base/manifests/init.pp
 create mode 100644 modules/thomas-base/spec/spec_helper.rb
 create mode 100644 modules/thomas-base/tests/init.pp
```

17. At this point your changes to the Git repository have been committed locally; you now need to push those changes back to `git.example.com` so that other nodes can retrieve the updated files:

```
t@mylaptop puppet$ git push origin master
Counting objects: 15, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (15/15), 2.15 KiB | 0 bytes/s, done.
Total 15 (delta 0), reused 0 (delta 0)
To git@git.example.com:repos/puppet.git
 * [new branch]      master -> master
```

How it works...

Git tracks changes to files, and stores a complete history of all changes. The history of the repo is made up of commits. A commit represents the state of the repo at a particular point in time, which you create with the `git commit` command and annotate with a message.

You've now added your Puppet manifest files to the repo and created your first commit. This updates the history of the repo, but only in your local working copy. To synchronize the changes with the `git.example.com` copy, the `git push` command pushes all changes made since the last sync.

There's more...

Now that you have a central Git repository for your Puppet manifests, you can check out multiple copies of it in different places and work on them before committing your changes. For example, if you're working in a team, each member can have their own local copy of the repo and synchronize changes with the others via the central server. You may also choose to use GitHub as your central Git repository server. GitHub offers free Git repository hosting for public repositories, and you can pay for GitHub's premium service if you don't want your Puppet code to be publicly available.

In the next section, we will use our Git repository for both centralized and decentralized Puppet configurations.

Creating a decentralized Puppet architecture

Puppet is a configuration management tool. You can use Puppet to configure and prevent configuration drift in a large number of client computers. If all your client computers are easily reached via a central location, you may choose to have a central Puppet server control all the client computers. In the centralized model, the Puppet server is known as the Puppet master. We will cover how to configure a central Puppet master in a few sections.

If your client computers are widely distributed or you cannot guarantee communication between the client computers and a central location, then a decentralized architecture may be a good fit for your deployment. In the next few sections, we will see how to configure a decentralized Puppet architecture.

As we have seen, we can run the `puppet apply` command directly on a manifest file to have Puppet apply it. The problem with this arrangement is that we need to have the manifests transferred to the client computers.

We can use the Git repository we created in the previous section to transfer our manifests to each new node we create.

Getting ready

Create a new test node, call this new node whatever you wish, I'll use `testnode` for mine. Install Puppet on the machine as we have previously done.

How to do it...

Create a `bootstrap.pp` manifest that will perform the following configuration steps on our new node:

1. Install Git:

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

2. Install the `ssh` key to access `git.example.com` in the Puppet user's home directory (`/var/lib/puppet/.ssh/id_rsa`):

```
File {
  owner => 'puppet',
  group => 'puppet',
}
file {'/var/lib/puppet/.ssh':
  ensure => 'directory',
}
file {'/var/lib/puppet/.ssh/id_rsa':
```

```
  content => "
-----BEGIN RSA PRIVATE KEY-----
…
NIjTXmZUlOKefh4MBilqUU3KQG8GBHjzYl2TkFVGLNYGNA0U8VG8SUJq
-----END RSA PRIVATE KEY-----
",
  mode    => 0600,
  require => File['/var/lib/puppet/.ssh']
}
```

3. Download the `ssh` host key from `git.example.com` (`/var/lib/puppet/.ssh/known_hosts`):

```
exec {'download git.example.com host key':
  command => 'sudo -u puppet ssh-keyscan git.example.com >> /var/
lib/puppet/.ssh/known_hosts',
  path    => '/usr/bin:/usr/sbin:/bin:/sbin',
  unless  => 'grep git.example.com /var/lib/puppet/.ssh/known_
hosts',
  require => File['/var/lib/puppet/.ssh'],
}
```

4. Create a directory to contain the Git repository (`/etc/puppet/cookbook`):

```
file {'/etc/puppet/cookbook':
  ensure => 'directory',
}
```

5. Clone the Puppet repository onto the new machine:

```
exec {'create cookbook':
  command => 'sudo -u puppet git clone git@git.example.com:repos/
puppet.git /etc/puppet/cookbook',
  path    => '/usr/bin:/usr/sbin:/bin:/sbin',
  require => [Package['git'],File['/var/lib/puppet/.ssh/id_
rsa'],Exec['download git.example.com host key']],
  unless  => 'test -f /etc/puppet/cookbook/.git/config',
}
```

6. Now when we run Puppet apply on the new machine, the `ssh` key will be installed for the Puppet user. The Puppet user will then clone the Git repository into `/etc/puppet/cookbook`:

```
root@testnode /tmp# puppet apply bootstrap.pp
Notice: Compiled catalog for testnode.example.com in environment
production in 0.40 seconds
Notice: /Stage[main]/Main/File[/etc/puppet/cookbook]/ensure:
created
Notice: /Stage[main]/Main/File[/var/lib/puppet/.ssh]/ensure:
created
```

```
Notice: /Stage[main]/Main/Exec[download git.example.com host key]/
returns: executed successfully
Notice: /Stage[main]/Main/File[/var/lib/puppet/.ssh/id_rsa]/
ensure: defined content as '{md5}da61ce6ccc79bc6937bd98c798bc9fd3'
Notice: /Stage[main]/Main/Exec[create cookbook]/returns: executed
successfully
Notice: Finished catalog run in 0.82 seconds
```

You may have to disable the `tty` requirement of `sudo`. Comment out the line `Defaults requiretty` at `/etc/sudoers` if you have this line.

Alternatively, you can set `user => Puppet` within the `'create cookbook' exec` type. Beware that using the user attribute will cause any error messages from the command to be lost.

7. Now that your Puppet code is available on the new node, you can apply it using `puppet apply`, specifying that `/etc/puppet/cookbook/modules` will contain the modules:

```
root@testnode ~# puppet apply --modulepath=/etc/puppet/cookbook/
modules /etc/puppet/cookbook/manifests/site.pp
Notice: Compiled catalog for testnode.example.com in environment
production in 0.12 seconds
Notice: /Stage[main]/Base/File[/etc/motd]/content: content changed
'{md5}86d28ff83a8d49d349ba56b5c64b79ee' to '{md5}4c4c3ab7591d94031
8279d78b9c51d4f'
Notice: Finished catalog run in 0.11 seconds
root@testnode /tmp# cat /etc/motd
testnode.example.com
Managed by puppet 3.6.2
```

How it works...

First, our `bootstrap.pp` manifest ensures that Git is installed. The manifest then goes on to ensure that the `ssh` key for the Git user on `git.example.com` is installed into the Puppet user's home directory (`/var/lib/puppet` by default). The manifest then ensures that the host key for `git.example.com` is trusted by the Puppet user. With `ssh` configured, the bootstrap ensures that `/etc/puppet/cookbook` exists and is a directory.

We then use an `exec` to have Git clone the repository into `/etc/puppet/cookbook`. With all the code in place, we then call `puppet apply` a final time to deploy the code from the repository. In a production setting, you would distribute the `bootstrap.pp` manifest to all your nodes, possibly via an internal web server, using a method similar to curl `http://puppet/bootstrap.pp >bootstrap.pp && puppet apply bootstrap.pp`

Writing a papply script

We'd like to make it as quick and easy as possible to apply Puppet on a machine; for this we'll write a little script that wraps the `puppet apply` command with the parameters it needs. We'll deploy the script where it's needed with Puppet itself.

How to do it...

Follow these steps:

1. In your Puppet repo, create the directories needed for a Puppet module:

```
t@mylaptop ~$ cd puppet/modules
t@mylaptop modules$ mkdir -p puppet/{manifests,files}
```

2. Create the `modules/puppet/files/papply.sh` file with the following contents:

```
#!/bin/sh
sudo puppet apply /etc/puppet/cookbook/manifests/site.pp \
  --modulepath=/etc/puppet/cookbook/modules $*
```

3. Create the `modules/puppet/manifests/init.pp` file with the following contents:

```
class puppet {
  file { '/usr/local/bin/papply':
    source => 'puppet:///modules/puppet/papply.sh',
    mode   => '0755',
  }
}
```

4. Modify your `manifests/site.pp` file as follows:

```
node default {
  include base
  include puppet
}
```

5. Add the Puppet module to the Git repository and commit the change as follows:

```
t@mylaptop puppet$ git add manifests/site.pp modules/puppet
t@mylaptop puppet$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
modified:   manifests/site.pp
```

```
new file:   modules/puppet/files/papply.sh
new file:   modules/puppet/manifests/init.pp
t@mylaptop puppet$ git commit -m "adding puppet module to include
papply"
[master 7c2e3d5] adding puppet module to include papply
 3 files changed, 11 insertions(+)
 create mode 100644 modules/puppet/files/papply.sh
 create mode 100644 modules/puppet/manifests/init.pp
```

6. Now remember to push the changes to the Git repository on `git.example.com`:

```
t@mylaptop puppet$ git push origin master
Counting objects: 14, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (10/10), 894 bytes | 0 bytes/s, done.
Total 10 (delta 0), reused 0 (delta 0)
To git@git.example.com:repos/puppet.git
   23e887c..7c2e3d5  master -> master
```

7. Pull the latest version of the Git repository to your new node (`testnode` for me) as shown in the following command line:

```
root@testnode ~# sudo -iu puppet
puppet@testnode ~$ cd /etc/puppet/cookbook/
puppet@testnode /etc/puppet/cookbook$ git pull origin master
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 10 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (10/10), done.
From git.example.com:repos/puppet
 * branch            master     -> FETCH_HEAD
Updating 23e887c..7c2e3d5
Fast-forward
 manifests/site.pp                |    1 +
 modules/puppet/files/papply.sh   |    4 ++++
 modules/puppet/manifests/init.pp |    6 ++++++
 3 files changed, 11 insertions(+), 0 deletions(-)
 create mode 100644 modules/puppet/files/papply.sh
 create mode 100644 modules/puppet/manifests/init.pp
```

8. Apply the manifest manually once to install the `papply` script:

```
root@testnode ~# puppet apply /etc/puppet/cookbook/manifests/site.
pp --modulepath /etc/puppet/cookbook/modules
Notice: Compiled catalog for testnode.example.com in environment
production in 0.13 seconds
```

```
Notice: /Stage[main]/Puppet/File[/usr/local/bin/papply]/ensure:
defined content as '{md5}d5c2cdd359306dd6e6441e6fb96e5ef7'
Notice: Finished catalog run in 0.13 seconds
```

9. Finally, test the script:

```
root@testnode ~# papply
Notice: Compiled catalog for testnode.example.com in environment
production in 0.13 seconds
Notice: Finished catalog run in 0.09 seconds
```

Now, whenever you need to run Puppet, you can simply run `papply`. In future, when we apply Puppet changes, I'll ask you to run `papply` instead of the full `puppet apply` command.

How it works...

As you've seen, to run Puppet on a machine and apply a specified manifest file, we use the `puppet apply` command:

```
puppet apply manifests/site.pp
```

When you're using modules (such as the Puppet module we just created), you also need to tell Puppet where to search for modules, using the `modulepath` argument:

```
puppet apply manifests/nodes.pp \
  --modulepath=/home/ubuntu/puppet/modules
```

In order to run Puppet with the root privileges it needs, we have to put `sudo` before everything:

```
sudo puppet apply manifests/nodes.pp \
  --modulepath=/home/ubuntu/puppet/modules
```

Finally, any additional arguments passed to `papply` will be passed through to Puppet itself, by adding the `$*` parameter:

```
sudo puppet apply manifests/nodes.pp \
  --modulepath=/home/ubuntu/puppet/modules $*
```

That's a lot of typing, so putting this in a script makes sense. We've added a Puppet file resource that will deploy the script to `/usr/local/bin` and make it executable:

```
file { '/usr/local/bin/papply':
  source => 'puppet:///modules/puppet/papply.sh',
  mode   => '0755',
}
```

Finally, we include the Puppet module in our default node declaration:

```
node default {
```

```
  include base
  include puppet
}
```

You can do the same for any other nodes managed by Puppet.

Running Puppet from cron

You can do a lot with the setup you already have: work on your Puppet manifests as a team, communicate changes via a central Git repository, and manually apply them on a machine using the `papply` script.

However, you still have to log into each machine to update the Git repo and rerun Puppet. It would be helpful to have each machine update itself and apply any changes automatically. Then all you need to do is to push a change to the repo, and it will go out to all your machines within a certain time.

The simplest way to do this is with a **cron** job that pulls updates from the repo at regular intervals and then runs Puppet if anything has changed.

Getting ready

You'll need the Git repo we set up in the *Managing your manifests with Git* and *Creating a decentralized Puppet architecture* recipes, and the `papply` script from the *Writing a papply script* recipe. You'll need to apply the `bootstrap.pp` manifest we created to install `ssh` keys to download the latest repository.

How to do it...

Follow these steps:

1. Copy the `bootstrap.pp` script to any node you wish to enroll. The `bootstrap.pp` manifest includes the private key used to access the Git repository, it should be protected in a production environment.
2. Create the `modules/puppet/files/pull-updates.sh` file with the following contents:

   ```
   #!/bin/sh
   cd /etc/puppet/cookbook
   sudo –u puppet git pull && /usr/local/bin/papply
   ```

3. Modify the `modules/puppet/manifests/init.pp` file and add the following snippet after the `papply` file definition:

   ```
   file { '/usr/local/bin/pull-updates':
     source => 'puppet:///modules/puppet/pull-updates.sh',
   ```

```
  mode   => '0755',
}
cron { 'run-puppet':
  ensure  => 'present',
  user    => 'puppet',
  command => '/usr/local/bin/pull-updates',
  minute  => '*/10',
  hour    => '*',
}
```

4. Commit the changes as before and push to the Git server as shown in the following command line:

```
t@mylaptop puppet$ git add modules/puppet
t@mylaptop puppet$ git commit -m "adding pull-updates"
[master 7e9bac3] adding pull-updates
 2 files changed, 14 insertions(+)
 create mode 100644 modules/puppet/files/pull-updates.sh
t@mylaptop puppet$ git push
Counting objects: 14, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (8/8), 839 bytes | 0 bytes/s, done.
Total 8 (delta 0), reused 0 (delta 0)
To git@git.example.com:repos/puppet.git
   7c2e3d5..7e9bac3  master -> master
```

5. Issue a Git pull on the test node:

```
root@testnode ~# cd /etc/puppet/cookbook/
root@testnode /etc/puppet/cookbook# sudo -u puppet git pull
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 8 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (8/8), done.
From git.example.com:repos/puppet
   23e887c..7e9bac3  master     -> origin/master
Updating 7c2e3d5..7e9bac3
Fast-forward
 modules/puppet/files/pull-updates.sh |    3 +++
 modules/puppet/manifests/init.pp     |   11 ++++++++++
 2 files changed, 14 insertions(+), 0 deletions(-)
 create mode 100644 modules/puppet/files/pull-updates.sh
```

6. Run Puppet on the test node:

```
root@testnode /etc/puppet/cookbook# papply
Notice: Compiled catalog for testnode.example.com in environment
production in 0.17 seconds
Notice: /Stage[main]/Puppet/Cron[run-puppet]/ensure: created
Notice: /Stage[main]/Puppet/File[/usr/local/bin/pull-updates]/
ensure: defined content as '{md5}04c023feb5d566a417b519ea51586398'
Notice: Finished catalog run in 0.16 seconds
```

7. Check that the `pull-updates` script works properly:

```
root@testnode /etc/puppet/cookbook# pull-updates
Already up-to-date.
Notice: Compiled catalog for testnode.example.com in environment
production in 0.15 seconds
Notice: Finished catalog run in 0.14 seconds
```

8. Verify the `cron` job was created successfully:

```
root@testnode /etc/puppet/cookbook# crontab -l -u puppet
# HEADER: This file was autogenerated at Tue Sep 09 02:31:00 -0400
2014 by puppet.
# HEADER: While it can still be managed manually, it is definitely
not recommended.
# HEADER: Note particularly that the comments starting with
'Puppet Name' should
# HEADER: not be deleted, as doing so could cause duplicate cron
jobs.
# Puppet Name: run-puppet
*/10 * * * * /usr/local/bin/pull-updates
```

How it works...

When we created the `bootstrap.pp` manifest, we made sure that the Puppet user can checkout the Git repository using an `ssh` key. This enables the Puppet user to run the Git pull in the cookbook directory unattended. We've also added the `pull-updates` script, which does this and runs Puppet if any changes are pulled:

```
#!/bin/sh
cd /etc/puppet/cookbook
sudo –u puppet git pull && papply
```

We deploy this script to the node with Puppet:

```
file { '/usr/local/bin/pull-updates':
  source => 'puppet:///modules/puppet/pull-updates.sh',
  mode   => '0755',
}
```

Finally, we've created a `cron` job that runs `pull-updates` at regular intervals (every 10 minutes, but feel free to change this if you need to):

```
cron { 'run-puppet':
  ensure  => 'present',
  command => '/usr/local/bin/pull-updates',
  minute  => '*/10',
  hour    => '*',
}
```

There's more...

Congratulations, you now have a fully-automated Puppet infrastructure! Once you have applied the `bootstrap.pp` manifest, run Puppet on the repository; the machine will be set up to pull any new changes and apply them automatically.

So, for example, if you wanted to add a new user account to all your machines, all you have to do is add the account in your working copy of the manifest, and commit and push the changes to the central Git repository. Within 10 minutes, it will automatically be applied to every machine that's running Puppet.

Bootstrapping Puppet with bash

Previous versions of this book used Rakefiles to bootstrap Puppet. The problem with using Rake to configure a node is that you are running the commands from your laptop; you assume you already have `ssh` access to the machine. Most bootstrap processes work by issuing an easy to remember command from a node once it has been provisioned. In this section, we'll show how to use bash to bootstrap Puppet with a web server and a bootstrap script.

Getting ready

Install httpd on a centrally accessible server and create a password protected area to store the bootstrap script. In my example, I'll use the Git server I set up previously, `git.example.com`. Start by creating a directory in the root of your web server:

```
# cd /var/www/html
# mkdir bootstrap
```

Now perform the following steps:

1. Add the following location definition to your apache configuration:

```
<Location /bootstrap>
AuthType basic
AuthName "Bootstrap"
AuthBasicProvider file
AuthUserFile /var/www/puppet.passwd
Require valid-user
</Location>
```

2. Reload your web server to ensure the location configuration is operating. Verify with curl that you cannot download from the bootstrap directory without authentication:

```
[root@bootstrap-test tmp]# curl http://git.example.com/bootstrap/
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>401 Authorization Required</title>
</head><body>
<h1>Authorization Required</h1>
```

3. Create the password file you referenced in the apache configuration (`/var/www/puppet.passwd`):

```
root@git# cd /var/www
root@git# htpasswd -cb puppet.passwd bootstrap cookbook
Adding password for user bootstrap
```

4. Verify that the username and password permit access to the bootstrap directory as follows:

```
[root@node1 tmp]# curl --user bootstrap:cookbook http://git.
example.com/bootstrap/
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>
 <head>
  <title>Index of /bootstrap</title>
```

How to do it...

Now that you have a safe location to store the bootstrap script, create a bootstrap script for each OS you support in the bootstrap directory. In this example, I'll show you how to do this for a Red Hat Enterprise Linux 6-based distribution.

Although the bootstrap location requires a password, there is no encryption since we haven't configured SSL on our server. Without encryption, the location is not very safe.

Create a script named `el6.sh` in the bootstrap directory with the following contents:

```
#!/bin/bash

# bootstrap for EL6 distributions
SERVER=git.example.com
LOCATION=/bootstrap
BOOTSTRAP=bootstrap.pp
USER=bootstrap
PASS=cookbook

# install puppet
curl http://yum.puppetlabs.com/RPM-GPG-KEY-puppetlabs >/etc/pki/rpm-
gpg/RPM-GPG-KEY-puppetlabs
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-puppetlabs
yum -y install http://yum.puppetlabs.com/puppetlabs-release-el-6.
noarch.rpm
yum -y install puppet
# download bootstrap
curl --user $USER:$PASS http://$SERVER/$LOCATION/$BOOTSTRAP >/
tmp/$BOOTSTRAP
# apply bootstrap
cd /tmp
puppet apply /tmp/$BOOTSTRAP
# apply puppet
puppet apply --modulepath /etc/puppet/cookbook/modules /etc/puppet/
cookbook/manifests/site.pp
```

How it works...

The apache configuration only permits access to the bootstrap directory with a username and password combination. We supply these with the `--user` argument to curl, thereby getting access to the file. We use a pipe (`|`) to redirect the output of curl into bash. This causes bash to execute the script. We write our bash script like we would any other bash script. The bash script downloads our `bootstrap.pp` manifest and applies it. Finally, we apply the Puppet manifest from the Git repository and the machine is configured as a member of our decentralized infrastructure.

There's more...

To support another operating system, we only need to create a new bash script. All Linux distributions will support bash scripting, Mac OS X does as well. Since we placed much of our logic into the `bootstrap.pp` manifest, the bootstrap script is quite minimal and easy to port to new operating systems.

Creating a centralized Puppet infrastructure

A configuration management tool such as Puppet is best used when you have many machines to manage. If all the machines can reach a central location, using a centralized Puppet infrastructure might be a good solution. Unfortunately, Puppet doesn't scale well with a large number of nodes. If your deployment has less than 800 servers, a single Puppet master should be able to handle the load, assuming your catalogs are not complex (take less than 10 seconds to compile each catalog). If you have a larger number of nodes, I suggest a load balancing configuration described in *Mastering Puppet*, *Thomas Uphill*, *Packt Publishing*.

A Puppet master is a Puppet server that acts as an X509 certificate authority for Puppet and distributes catalogs (compiled manifests) to client nodes. Puppet ships with a built-in web server called **WEBrick,** which can handle a very small number of nodes. In this section, we will see how to use that built-in server to control a very small (less than 10) number of nodes.

Getting ready

The Puppet master process is started by running `puppet master`; most Linux distributions have start and stop scripts for the Puppet master in a separate package. To get started, we'll create a new debian server named `puppet.example.com`.

How to do it...

1. Install Puppet on the new server and then use Puppet to install the Puppet master package:

```
# puppet resource package puppetmaster ensure='installed'
Notice: /Package[puppetmaster]/ensure: created
package { 'puppetmaster':
  ensure => '3.7.0-1puppetlabs1',
}
```

2. Now start the Puppet master service and ensure it will start at boot:

```
# puppet resource service puppetmaster ensure=true enable=true
service { 'puppetmaster':
  ensure => 'running',
  enable => 'true',
}
```

How it works...

The Puppet master package includes the start and stop scripts for the Puppet master service. We use Puppet to install the package and start the service. Once the service is started, we can point another node at the Puppet master (you might need to disable the host-based firewall on your machine).

1. From another node, run `puppet agent` to start a `puppet agent`, which will contact the server and request a new certificate:

```
t@ckbk:~$ sudo puppet agent -t
Info: Creating a new SSL key for cookbook.example.com
Info: Caching certificate for ca
Info: Creating a new SSL certificate request for cookbook.example.
com
Info: Certificate Request fingerprint (SHA256): 06:C6:2B:C4:97:5D:
16:F2:73:82:C4:A9:A7:B1:D0:95:AC:69:7B:27:13:A9:1A:4C:98:20:21:C2:
50:48:66:A2
Info: Caching certificate for ca
Exiting; no certificate found and waitforcert is disabled
```

2. Now on the Puppet server, sign the new key:

```
root@puppet:~# puppet cert list
pu  "cookbook.example.com" (SHA256) 06:C6:2B:C4:97:5D:16:F2:73:82:
C4:A9:A7:B1:D0:95:AC:69:7B:27:13:A9:1A:4C:98:20:21:C2:50:48:66:A2
root@puppet:~# puppet cert sign cookbook.example.com
Notice: Signed certificate request for cookbook.example.com
Notice: Removing file Puppet::SSL::CertificateRequest
cookbook.example.com at
'/var/lib/puppet/ssl/ca/requests/cookbook.example.com.pem'
```

3. Return to the cookbook node and run Puppet again:

```
t@ckbk:~$ sudo puppet agent –vt
Info: Caching certificate for cookbook.example.com
Info: Caching certificate_revocation_list for ca
Info: Caching certificate for cookbook.example.comInfo: Retrieving
pluginfacts
Info: Retrieving plugin
Info: Caching catalog for cookbook
Info: Applying configuration version '1410401823'
Notice: Finished catalog run in 0.04 seconds
```

There's more...

When we ran `puppet agent`, Puppet looked for a host named `puppet.example.com` (since our test node is in the `example.com` domain); if it couldn't find that host, it would then look for a host named Puppet. We can specify the server to contact with the `--server` option to `puppet agent`. When we installed the Puppet master package and started the Puppet master service, Puppet created default SSL certificates based on our hostname. In the next section, we'll see how to create an SSL certificate that has multiple DNS names for our Puppet server.

Creating certificates with multiple DNS names

By default, Puppet will create an SSL certificate for your Puppet master that contains the fully qualified domain name of the server only. Depending on how your network is configured, it can be useful for the server to be known by other names. In this recipe, we'll make a new certificate for our Puppet master that has multiple DNS names.

Getting ready

Install the Puppet master package if you haven't already done so. You will then need to start the Puppet master service at least once to create a **certificate authority** (**CA**).

How to do it...

The steps are as follows:

1. Stop the running Puppet master process with the following command:

```
# service puppetmaster stop
[ ok ] Stopping puppet master.
```

2. Delete (`clean`) the current server certificate:

```
# puppet cert clean puppet
Notice: Revoked certificate with serial 6
Notice: Removing file Puppet::SSL::Certificate puppet at '/var/
lib/puppet/ssl/ca/signed/puppet.pem'
Notice: Removing file Puppet::SSL::Certificate puppet at '/var/
lib/puppet/ssl/certs/puppet.pem'
Notice: Removing file Puppet::SSL::Key puppet at '/var/lib/puppet/
ssl/private_keys/puppet.pem'
```

3. Create a new Puppet certificate using Puppet certificate generate with the `--dns-alt-names` option:

```
root@puppet:~# puppet certificate generate puppet --dns-alt-names
puppet.example.com,puppet.example.org,puppet.example.net --ca-
location local
Notice: puppet has a waiting certificate request
true
```

4. Sign the new certificate:

```
root@puppet:~# puppet cert --allow-dns-alt-names sign puppet
Notice: Signed certificate request for puppet
Notice: Removing file Puppet::SSL::CertificateRequest puppet at '/
var/lib/puppet/ssl/ca/requests/puppet.pem'
```

5. Restart the Puppet master process:

```
root@puppet:~# service puppetmaster restart
[ ok ] Restarting puppet master.
```

How it works...

When your puppet agents connect to the Puppet server, they look for a host called `Puppet`, they then look for a host called `Puppet.[your domain]`. If your clients are in different domains, then you need your Puppet master to reply to all the names correctly. By removing the existing certificate and generating a new one, you can have your Puppet master reply to multiple DNS names.

Running Puppet from passenger

The WEBrick server we configured in the previous section is not capable of handling a large number of nodes. To deal with a large number of nodes, a scalable web server is required. Puppet is a ruby process, so we need a way to run a ruby process within a web server. **Passenger** is the solution to this problem. It allows us to run the Puppet master process within a web server (apache by default). Many distributions ship with a puppetmaster-passenger package that configures this for you. In this section, we'll use the package to configure Puppet to run within passenger.

Getting ready

Install the puppetmaster-passenger package:

```
# puppet resource package puppetmaster-passenger ensure=installed
Notice: /Package[puppetmaster-passenger]/ensure: ensure changed 'purged'
```

```
 to 'present'
package { 'puppetmaster-passenger':
  ensure => '3.7.0-1puppetlabs1',
}
```

Using `puppet resource` to install packages ensures the same command will work on multiple distributions (provided the package names are the same).

How to do it...

The steps are as follows:

1. Ensure the Puppet master site is enabled in your apache configuration. Depending on your distribution this may be at `/etc/httpd/conf.d` or `/etc/apache2/sites-enabled`. The configuration file should be created for you and contain the following information:

```
PassengerHighPerformance on
PassengerMaxPoolSize 12
PassengerPoolIdleTime 1500
# PassengerMaxRequests 1000
PassengerStatThrottleRate 120
RackAutoDetect Off
RailsAutoDetect Off
Listen 8140
```

2. These lines are tuning settings for passenger. The file then instructs apache to listen on port 8140, the Puppet master port. Next a `VirtualHost` definition is created that loads the Puppet CA certificates and the Puppet master's certificate:

```
<VirtualHost *:8140>
        SSLEngine on
        SSLProtocol             ALL -SSLv2 -SSLv3
        SSLCertificateFile      /var/lib/puppet/ssl/certs/puppet.
pem
        SSLCertificateKeyFile   /var/lib/puppet/ssl/private_keys/
puppet.pem
        SSLCertificateChainFile /var/lib/puppet/ssl/certs/ca.pem
        SSLCACertificateFile    /var/lib/puppet/ssl/certs/ca.pem
        SSLCARevocationFile     /var/lib/puppet/ssl/ca/ca_crl.pem
```

```
SSLVerifyClient optional
SSLVerifyDepth  1
SSLOptions +StdEnvVars +ExportCertData
```

You may have more or less lines of SSL configuration here depending on your version of the puppetmaster-passenger package.

3. Next, a few important headers are set so that the passenger process has access to the SSL information sent by the client node:

```
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
```

4. Finally, the location of the passenger configuration file `config.ru` is given with the `DocumentRoot` location as follows:

```
DocumentRoot /usr/share/puppet/rack/puppetmasterd/public/
 RackBaseURI /
```

5. The `config.ru` file should exist at `/usr/share/puppet/rack/puppetmasterd/` and should have the following content:

```
$0 = "master"
ARGV << "--rack"
ARGV << "--confdir" << "/etc/puppet"
ARGV << "--vardir"  << "/var/lib/puppet"
require 'puppet/util/command_line'
run Puppet::Util::CommandLine.new.execute
```

6. With the passenger apache configuration file in place and the `config.ru` file correctly configured, start the apache server and verify that apache is listening on the Puppet master port (if you configured the standalone Puppet master previously, you must stop that process now using `service puppetmaster stop`):

```
root@puppet:~ # service apache2 start
[ ok ] Starting web server: apache2
root@puppet:~ # lsof -i :8140
COMMAND  PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
apache2 9048     root    8u  IPv6  16842      0t0  TCP *:8140
(LISTEN)
```

```
apache2 9069 www-data    8u  IPv6  16842      0t0  TCP *:8140
(LISTEN)
apache2 9070 www-data    8u  IPv6  16842      0t0  TCP *:8140
(LISTEN)
```

How it works...

The passenger configuration file uses the existing Puppet master certificates to listen on port 8140 and handles all the SSL communication between the server and the client. Once the certificate information has been dealt with, the connection is handed off to a ruby process started from passenger using the command line arguments from the `config.ru` file.

In this case, the `$0` variable is set to `master` and the arguments variable is set to `--rack --confdir /etc/puppet --vardir /var/lib/puppet`; this is equivalent to running the following from the command line:

```
puppet master --rack --confdir /etc/puppet --vardir /var/lib/puppet
```

There's more...

You can add additional configuration parameters to the `config.ru` file to further alter how Puppet runs when it's running through passenger. For instance, to enable debugging on the passenger Puppet master, add the following line to `config.ru` before the run `Puppet::Util::CommandLine.new.execute` line:

```
ARGV << "--debug"
```

Setting up the environment

Environments in Puppet are directories holding different versions of your Puppet manifests. Environments prior to Version 3.6 of Puppet were not a default configuration for Puppet. In newer versions of Puppet, environments are configured by default.

Whenever a node connects to a Puppet master, it informs the Puppet master of its environment. By default, all nodes report to the `production` environment. This causes the Puppet master to look in the production environment for manifests. You may specify an alternate environment with the `--environment` setting when running puppet agent or by setting environment `= newenvironment` in `/etc/puppet/puppet.conf` in the [agent] section.

Getting ready

Set the `environmentpath` function of your installation by adding a line to the `[main]` section of `/etc/puppet/puppet.conf` as follows:

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

How to do it...

The steps are as follows:

1. Create a `production` directory at `/etc/puppet/environments` that contains both a `modules` and `manifests` directory. Then create a `site.pp` which creates a file in `/tmp` as follows:

```
root@puppet:~# cd /etc/puppet/environments/
root@puppet:/etc/puppet/environments# mkdir -p production/
{manifests,modules}
root@puppet:/etc/puppet/environments# vim production/manifests/
site.pp
node default {
  file {'/tmp/production':
    content => "Hello World!\nThis is production\n",
  }
}
```

2. Run puppet agent on the master to connect to it and verify that the production code was delivered:

```
root@puppet:~# puppet agent -vt
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for puppet
Info: Applying configuration version '1410415538'
Notice: /Stage[main]/Main/Node[default]/File[/tmp/production]/
ensure: defined content as '{md5}f7ad9261670b9da33a67a5126933044c'
Notice: Finished catalog run in 0.04 seconds
# cat /tmp/production
Hello World!
This is production
```

3. Configure another environment `devel`. Create a new manifest in the `devel` environment:

```
root@puppet:/etc/puppet/environments# mkdir -p devel/
{manifests,modules}
root@puppet:/etc/puppet/environments# vim devel/manifests/site.pp
node default {
  file {'/tmp/devel':
    content => "Good-bye! Development\n",
  }
}
```

4. Apply the new environment by running the `--environment devel` puppet agent using the following command:

```
root@puppet:/etc/puppet/environments# puppet agent -vt
--environment devel
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for puppet
Info: Applying configuration version '1410415890'
Notice: /Stage[main]/Main/Node[default]/File[/tmp/devel]/ensure:
defined content as '{md5}b6313bb89bc1b7d97eae5aa94588eb68'
Notice: Finished catalog run in 0.04 seconds
root@puppet:/etc/puppet/environments# cat /tmp/devel
Good-bye! Development
```

You may need to restart apache2 to enable your new environment, this depends on your version of Puppet and the `environment_timeout` parameter of `puppet.conf`.

There's more...

Each environment can have its own `modulepath` if you create an `environment.conf` file within the environment directory. More information on environments can be found on the Puppet labs website at `https://docs.puppetlabs.com/puppet/latest/reference/environments.html`.

Configuring PuppetDB

PuppetDB is a database for Puppet that is used to store information about nodes connected to a Puppet master. PuppetDB is also a storage area for exported resources. Exported resources are resources that are defined on nodes but applied to other nodes. The simplest way to install PuppetDB is to use the PuppetDB module from Puppet labs. From this point on, we'll assume you are using the `puppet.example.com` machine and have a passenger-based configuration of Puppet.

Getting ready

Install the PuppetDB module in the production environment you created in the previous recipe. If you didn't create directory environments, don't worry, using `puppet module install` will install the module to the correct location for your installation with the following command:

```
root@puppet:~# puppet module install puppetlabs-puppetdb
Notice: Preparing to install into /etc/puppet/environments/production/
modules ...
Notice: Downloading from https://forgeapi.puppetlabs.com ...
Notice: Installing -- do not interrupt ...
/etc/puppet/environments/production/modules
└─┬ puppetlabs-puppetdb (v3.0.1)
  ├── puppetlabs-firewall (v1.1.3)
  ├── puppetlabs-inifile (v1.1.3)
  └─┬ puppetlabs-postgresql (v3.4.2)
    ├─┬ puppetlabs-apt (v1.6.0)
    │ └── puppetlabs-stdlib (v4.3.2)
    └── puppetlabs-concat (v1.1.0)
```

How to do it...

Now that our Puppet master has the PuppetDB module installed, we need to apply the PuppetDB module to our Puppet master, we can do this in the site manifest. Add the following to your (production) `site.pp`:

```
node puppet {
  class { 'puppetdb': }
  class { 'puppetdb::master::config':
    puppet_service_name => 'apache2',
  }
}
```

Run `puppet agent` to apply the `puppetdb` class and the `puppetdb::master::config` class:

```
root@puppet:~# puppet agent -t
Info: Caching catalog for puppet
Info: Applying configuration version '1410416952'
...
Info: Class[Puppetdb::Server::Jetty_ini]: Scheduling refresh of
Service[puppetdb]
Notice: Finished catalog run in 160.78 seconds
```

How it works...

The PuppetDB module is a great example of how a complex configuration task can be puppetized. Simply by adding the `puppetdb` class to our Puppet master node, Puppet installed and configured `postgresql` and `puppetdb`.

When we called the `puppetdb::master::config` class, we set the `puppet_service_name` variable to `apache2`, this is because we are running Puppet through passenger. Without this line our agent would try to start the puppetmaster process instead of `apache2`.

The agent then set up the configuration files for PuppetDB and configured Puppet to use PuppetDB. If you look at `/etc/puppet/puppet.conf`, you'll see the following two new lines:

```
storeconfigs = true
storeconfigs_backend = puppetdb
```

There's more...

Now that PuppetDB is configured and we've had a successful agent run, PuppetDB will have data we can query:

```
root@puppet:~# puppet node status puppet
puppet
Currently active
Last catalog: 2014-09-11T06:45:25.267Z
Last facts: 2014-09-11T06:45:22.351Z
```

Configuring Hiera

Hiera is an information repository for Puppet. Using Hiera you can have a hierarchical categorization of data about your nodes that is maintained outside of your manifests. This is very useful for sharing code and dealing with exceptions that will creep into any Puppet deployment.

Getting ready

Hiera should have already been installed as a dependency on your Puppet master. If it has not already, install it using Puppet:

```
root@puppet:~# puppet resource package hiera ensure=installed
package { 'hiera':
  ensure => '1.3.4-1puppetlabs1',
}
```

How to do it...

1. Hiera is configured from a yaml file, `/etc/puppet/hiera.yaml`. Create the file and add the following as a minimal configuration:

```
---
:hierarchy:
  - common
:backends:
  - yaml
:yaml:
  :datadir: '/etc/puppet/hieradata'
```

2. Create the `common.yaml` file referenced in the hierarchy:

```
root@puppet:/etc/puppet# mkdir hieradata
root@puppet:/etc/puppet# vim hieradata/common.yaml
---
message: 'Default Message'
```

3. Edit the `site.pp` file and add a notify resource based on the Hiera value:

```
node default {
  $message = hiera('message','unknown')
  notify {"Message is $message":}
}
```

4. Apply the manifest to a test node:

```
t@ckbk:~$ sudo puppet agent -t
Info: Retrieving pluginfacts
Info: Retrieving plugin
...
Info: Caching catalog for cookbook-test
Info: Applying configuration version '1410504848'
Notice: Message is Default Message
```