Quick Answers to common problems

Puppet 2.7 Cookbook: RAW

Build reliable, scalable, secure, high-performance systems to fully utilize the power of cloud computing

John Arundel

[PACKT] open source* community experience distilled PUBLISHING

Puppet 2.7 Cookbook

RAW Book

Build reliable, scalable, secure, high-performance systems to fully utilize the power of cloud computing

John Arundel

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Puppet 2.7 Cookbook

Current RAW Publication: July 2011

RAW Production Reference: 1050711

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849515-38-2

`www.packtpub.com`

About the Author

John Arundel is a sysadmin, architect and systems integrator of 20 years experience. He is a published author of several technical books, a well-known expert on Puppet, the open-source configuration management system, and speaks regularly at technical user groups and conferences. Together with a small network of trusted associates, John has been helping small businesses grow for over five years.

Table of Contents

Preface 1

Chapter 1: Puppet Infrastructure 7

Introduction 7
Using version control 8
Deploying changes with Rake 10
Configuring Puppet's file server 13
Running Puppet from cron 15
Retrieving files from Puppet's filebucket 16
Using autosign 18
Pre-signing certificates 19
Scaling Puppet using Passenger 20
Creating a decentralized Puppet architecture 24

Chapter 2: Monitoring, Reporting, and Troubleshooting 29

Introduction 30
Generating reports 30
Emailing log messages containing 32
specific tags 32
Creating graphical reports 34
Producing automatic HTML documentation 34
Drawing dependency graphs 36
Testing your Puppet manifests 39
Doing a dry run 42
Detecting compilation errors 44
Understanding Puppet errors 45
Logging command output 47
Logging debug messages 48
Inspecting configuration settings 49
Using tags 50

Using run stages 52
Using environments 54
Chapter 3: Puppet Language and Style 57
Using modules 58
Using standard naming conventions 61
Using community Puppet style 63
Using embedded Ruby 65
Writing manifests in pure Ruby 65
Iterating over multiple items 68
Writing powerful conditional statements 70
Using regular expressions in `if` statements 72
Using selectors and case statements 74
Checking if values are contained in strings 76
Using regular expression substitutions 77
Chapter 4: Writing Better Manifests 79
Introduction 80
Using arrays of resources 80
Using defines 81
Using dependencies 83
Using node inheritance 86
Using class inheritance and overriding 88
Passing parameters to classes 93
Writing reusable, cross-platform manifests 95
Getting information about the environment 97
Importing dynamic information with generate 98
Importing data from CSV files 100
Passing arguments to shell commands 103
Chapter 5: Working with Files and Packages 105
Making quick edits to config files 106
Using Augeas to automatically edit config files 107
Building config files using snippets 110
Using ERB templates 112
Using array iteration in templates 116
Installing packages from a third-party repository 118
Setting up an APT package repository 119
Setting up a gem repository 125
Building packages automatically from source 128
Comparing package versions 130

Chapter 6: Users and Other Resources 133
Using virtual resources 134
Managing users with virtual resources 136
Managing users' SSH access 139
Managing users' customization files 142
Efficiently distributing cron jobs 144
Running a command when a file is updated 145
Using host resources 149
Using multiple file sources 150
Distributing directory trees using recursive file resources 153
Cleaning up old files 155
Using schedules with resources 156
Auditing resources 158
Temporarily disabling resources 159
Managing timezones 161

Preface

Welcome to Puppet 2.7 Cookbook, the RAW edition. A RAW (Read As we Write) book contains all the material written for the book so far, but available for you right now, before it's finished. As the author writes more, you will be invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW!

Ever wanted a robot butler? It may not bring you drinks, but Puppet can do most everything else. System administration used to be dominated by tedious, manual, daily operations tasks and repetitive, error-prone activities such as building servers and installing software. The advent of configuration management systems - robot sysadmins - has helped to automate away much of the manual work and freed us humans to solve more interesting problems such as:

- How can servers provision themselves, install their own software stack, deploy their own applications, and monitor themselves, all without your intervention?
- How can you keep all your servers identical, even when dependencies change?
- How can you build dynamically-scalable architectures which automatically add or reduce capacity in response to demand?
- How can you effectively co-ordinate and control changes made by distributed teams of developers and sysadmins?
- How can you build your infrastructure from simple, easy-to-read specifications that are always up-to-date?
- How can you automate manual processes, reduce maintenance and downtime, and give yourself the time to address performance improvements, capacity planning, long-term strategy, and business focus?
- *How can you be agile in responding to changing customer demand, changing developer requirements, and changing business needs?*

Answers to all of these questions can be found in this book. You'll learn to take Puppet to the limits of its capabilities, explore the latest features, and implement cutting-edge ideas from some of today's leading Puppet experts. Each recipe contains valuable, real-life examples that you can use in production on your own systems, and shows you how to apply and adapt them with step-by-step instructions and console commands.

The book is structured so that you can dip in at any point and try out a recipe without having to work your way through from cover to cover. You'll find links and references to more information on every topic, so that you can explore further for yourself. Whatever your level of Puppet experience, there's something for you, from simple workflow tips to advanced, high-performance Puppet architectures.

I've tried hard to write the kind of book that would be useful in your everyday work as an infrastructure engineer or consultant. I hope it will inspire you to learn, experiment, and to come up with your own new ideas in this exciting and fast-moving field.

What's in This RAW Book

In this RAW book, you will find these chapters:

Chapter 1: *Puppet infrastructure*, In this chapter you will understand how to implement the best methods for using version control with Puppet, automating your workflow, configuring Puppet for file serving, certificate management, improving Puppetmaster performance, and building a distributed Puppet architecture based on Git.

Chapter 2: *Monitoring, reporting and troubleshooting*, Through this chapter you will learn to use environments, tags, and run stages for generating reports and documentation with Puppet, testing and dry-running your manifests. It will help you understand Puppet's error messages and logging debug information.

Chapter 3: *Puppet language and style*, In this chapter you will learn to structure your manifests with modules, maintain your code using standard naming and style conventions, build config files dynamically and iterate over arrays using embedded Ruby in your templates. Finally you will learn to perform different computations using iteration, conditionals, and regular expressions, using selectors and case statements.

Chapter 4: *Writing better manifests*, This chapter builds on the concepts introduced in Chapter 3 teaching you to maintain and organize your code by using arrays, ordering your resources with dependencies, using class and node inheritance and overriding parameters. You will also learn how to read information from the environment, import data from external scripts and CSV files, and discover how to to write reusable manifests.

Chapter 5: *Working with files and packages*,Through this chapter you will learn how to manage config files, use Augeas to automatically edit config files, generate files from templates, manage third-party package repositories, build your own package and gem repositories, and build packages from its source.

Chapter 6: *Users and other resources*: In this chapter you will understand how to simplify the administration of a large number of users using virtual resources, manage their SSH keys and customization files, distribute directory trees and cron jobs, use multiple file sources and host resources, and also how to clean up files with tidy resources.

What's Still to Come?

We mentioned before that the book isn't finished, and here is what we currently plan to include in the remainder of the book:

Chapter 7: Applications: Learn to work with popular application platforms such as Drupal and Rails to manage your mailing lists and source code repositories with the help of MySQL, Apache, Nginx, and Tomcat.

Chapter 8: Server and cloud infrastructure: Through this chapter you will learn to work with EC2, Rackspace, and similar other cloud platforms. It also covers load balancing and virtual IPs, NFS file servers, logical volume managers, OpenVZ virtual machines, and Windows.

Chapter 9: External tools and the Puppet ecosystem, In this chapter you will discover what you can do to bring out the best from Puppet using external tools such as Dashboard ,Foreman, and MCollective. This chapter also teaches you how to create Facter facts, automatically generate Puppet manifests, download publically-available modules, work with external node classifiers, and create your own resource types and providers.

Conventions

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

Code words in text are shown as follows: "Puppet's `regsubst` function provides an easy way to manipulate text."

A block of code is set as follows:

```
$class_c = regsubst($ipaddress, "(.*)\..*", "\1.0")
notify { $ipaddress: }
notify { $class_c: }
```

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

```
$class_c = regsubst($ipaddress, "(.*)\..*", "\1.0")
notify { $ipaddress: }
notify { $class_c: }
```

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

```
# puppet agent –test
```

```
notice: The latest stable Puppet version is 2.6.7. You're using 2.6.4.
```

New terms and important words are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the Next button moves you to the next screen".

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

> Tips and tricks appear like this.

What Is a RAW Book?

Buying a Packt RAW book allows you to access Packt books before they're published. A RAW (Read As we Write) book is an eBook available for immediate download, and containing all the material written for the book so far.

As the author writes more, you are invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW! With a RAW book, you get immediate access, and the opportunity to participate in the development of the book, making sure that your voice is heard to get the kind of book that you want.

Is a RAW Book a Proper Book?

Yes, but it's just not all there yet! RAW chapters will be released as soon as we are happy for them to go into your book—we want you to have material that you can read and use straightaway. However, they will not have been through the full editorial process yet. You are receiving RAW content, available as soon as it written. If you find errors or mistakes in the book, or you think there are things that could be done better, you can contact us and we will make sure to get these things right before the final version is published.

When Do Chapters Become Available?

As soon as a chapter has been written and we are happy for it go into the RAW book, the new chapter will be added into the RAW eBook in your account. You will be notified that another chapter has become available and be invited to download it from your account. eBooks are licensed to you only; however, you are entitled to download them as often as you like and on as many different computers as you wish.

How Do I Know When New Chapters Are Released?

When new chapters are released all RAW customers will be notified by email with instructions on how to download their new eBook. Packt will also update the book's page on its website with a list of the available chapters.

Where Do I Get the Book From?

You download your RAW book much in the same way as any Packt eBook. In the download area of your Packt account, you will have a link to download the RAW book.

What Happens If I Have Problems with My RAW Book?

You are a Packt customer and as such, will be able to contact our dedicated Customer Service team. Therefore, if you experience any problems opening or downloading your RAW book, contact `service@packtpub.com` and they will reply to you quickly and courteously as they would to any Packt customer.

Is There Source Code Available During the RAW Phase?

Any source code for the RAW book can be downloaded from the Support page of our website (`http://www.packtpub.com/support`). Simply select the book from the list.

How Do I Post Feedback and Errata for a RAW Title?

If you find mistakes in this book, or things that you can think can be done better, let us know. You can contact us directly at `rawbooks@packtpub.com` to discuss any concerns you may have with the book.

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of. To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

1
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:

- Using version control
- Deploying changes with Rake
- Configuring Puppet's file server
- Running Puppet from `cron`
- Retrieving files from Puppet's filebucket
- Using `autosign`
- Pre-signing certificates
- Scaling Puppet using Passenger
- Creating a decentralized Puppet architecture

Introduction

Some of the recipes in this chapter, and in the rest of this book, represent best practice, agreed on by the Puppet community in general. Others are tips and tricks which will make it easier for you to work with Puppet, or introduce you to features which you may not have been previously aware of. Some recipes are short-cuts which I wouldn't recommend you use as standard operating procedure, but may be useful in emergencies. Finally, there are some experimental recipes which you may like to try, but are only useful or applicable in very large infrastructures or otherwise unusual circumstances.

My hope is that, by reading through and thinking about the recipes presented here, you will gain a deeper and broader understanding of how Puppet works and how you can use it to help you build better infrastructures. Only you can decide whether a particular recipe is appropriate for you and your organization, but I hope this collection will inspire you to experiment, find out more, and most of all - have fun!

Using version control

Ever deleted something and wished you hadn't? The most important tip in this book is to put your Puppet manifests in a **version control system** such as Git or Subversion. Editing the manifests directly on the Puppetmaster is a bad idea, because your changes could get applied before you're ready. Puppet automatically detects any changes to manifest files, so you might find half-finished manifests being applied to your clients. This could have nasty results!

Instead, use version control (I recommend Git) and make the `/etc/puppet` directory on the Puppetmaster a checkout from your repository. This gives you several advantages:

- You don't run the risk of Puppet applying incomplete changes
- You can undo changes and roll back to any previous version of your manifest
- You can experiment with new features using a branch, without affecting the master version used in production
- 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 log feature to see what was changed, and when (and by whom)

Getting ready

You'll need a Puppetmaster and a set of existing manifests in `/etc/puppet`. If you don't have these already, refer to the Puppet documentation to find out how to install Puppet and create your first manifests.

To put your manifests under version control, you're going to import the `/etc/puppet` directory from the Puppetmaster into your version control system, and make it a working copy. In this example, we'll use Git.

How to do it...

1. Turn the `/etc/puppet` directory on the Puppetmaster into a Git repository:

```
root@cookbook:/etc/puppet# git init
Initialized empty Git repository in /etc/puppet/.git/
root@cookbook:/etc/puppet# git add *
```

```
root@cookbook:/etc/puppet# git commit -m "initial commit"
[master (root-commit) 26d668c] initial commit
 2 files changed, 104 insertions(+), 0 deletions(-)
 create mode 100644 auth.conf
 create mode 100644 puppet.conf
```

2. Clone this to a bare Git repository:

```
root@cookbook:/etc/puppet# git clone --bare /etc/puppet /var/git/puppet.git
Initialized empty Git repository in /var/git/puppet.git/
```

3. Add the bare repo as the origin for the `/etc/puppet` checkout:

```
root@cookbook:/etc/puppet# git remote add -t master origin /var/git/puppet.git
```

3. Create a separate checkout for you to work in, so that you are not editing the one used by the Puppetmaster:

```
root@cookbook:/etc/puppet# cd
root@cookbook:~# git clone /var/git/puppet.git puppet-work
Initialized empty Git repository in /root/puppet-work/.git/
```

How it works...

You've created a master repo which you can check out anywhere and work on, before committing your changes. The Puppetmaster has its own checkout which you can update with the latest version from the master repo, once you are satisfied that it's ready for production.

There's more...

Now that you've set up version control, you can use the following workflow for editing your Puppet manifests:

1. Make your changes in the working copy
2. Commit the changes and push them to the origin repo
3. Update the Puppetmaster's working copy

Here is an example where we add a new file to the manifest, commit it, and then update the Puppetmaster's working copy:

```
root@cookbook:~# cd puppet-work
root@cookbook:~/puppet-work# mkdir manifests
root@cookbook:~/puppet-work# touch manifests/nodes.pp
```

```
root@cookbook:~/puppet-work# git add manifests/nodes.pp
root@cookbook:~/puppet-work# git commit -m "adding nodes.pp"
        [master 5c7b94c] adding nodes.pp
        0 files changed, 0 insertions(+), 0 deletions(-)
         create mode 100644 manifests/nodes.pp
root@cookbook:~/puppet-work# git push
        Counting objects: 5, done.
        Compressing objects: 100% (2/2), done.
        Writing objects: 100% (4/4), 365 bytes, done.
        Total 4 (delta 0), reused 0 (delta 0)
        Unpacking objects: 100% (4/4), done.
        To /var/git/puppet.git
           26d668c..5c7b94c  master -> master
root@cookbook:~/puppet-work# cd /etc/puppet
root@cookbook:/etc/puppet# git pull
        remote: Counting objects: 5, done.
        remote: Compressing objects: 100% (2/2), done.
        remote: Total 4 (delta 0), reused 0 (delta 0)
        Unpacking objects: 100% (4/4), done.
        From /var/git/puppet
          26d668c..5c7b94c  master     -> origin/master
        Updating 26d668c..5c7b94c
        Fast-forward
        0 files changed, 0 insertions(+), 0 deletions(-)
         create mode 100644 manifests/nodes.pp
```

You can automate this process by using a tool such as Rake.

See also

- Deploying changes with Rake
- Creating a decentralized Puppet architecture
- Using commit hooks

Deploying changes with Rake

Like everyone who makes his living with a keyboard, I hate unnecessary typing. If you are using the workflow described in 'Using version control for your Puppet manifests', you can add some automation to make this process a little easier. There are several tools which can run commands for you on remote servers, including Capistrano and Fabric, but for this example we'll use Rake.

Getting ready

If you don't have the Rake gem installed already, run:

```
gem install rake
```

How to do it...

1. Create a file in the top level of your Puppet working copy named `Rakefile` that looks like this:

   ```
   PUPPETMASTER = 'cookbook'
   SSH = 'ssh -t -A'

   task :deploy do
       sh "git push"
       sh "#{SSH} #{PUPPETMASTER} 'cd /etc/puppet && sudo git pull'"
   end
   ```

2. When you make changes in your working copy of the Puppet manifests, you can simply run:

   ```
   rake deploy
   ```

and Rake will take care of updating the Git repo and refreshing the Puppetmaster's working copy for you:

```
git push
Counting objects: 4, done.
Delta compression using 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 452 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@cookbook.bitfieldconsulting.com/var/git/cookbook
   561e5a6..a8b8c76  master -> master
ssh -A -l root cookbook 'cd /etc/puppet && git pull'
From ssh://cookbook.bitfieldconsulting.com/var/git/cookbook
   561e5a6..a8b8c76  master     -> origin/master
Updating 561e5a6..a8b8c76
Fast-forward
Rakefile |    6 ++++++
1 files changed, 6 insertions(+), 0 deletions(-)
create mode 100644 Rakefile
```

3. You can also add a Rake task to run Puppet on the client machine:

```
task :apply => [:deploy] do
    client = ENV['CLIENT']
    sh "#{SSH} #{client} 'sudo puppet agent --test'" do |ok,
status|
        puts case status.exitstatus
            when 0 then "Client is up to date."
            when 1 then "Puppet couldn't compile the manifest."
            when 2 then "Puppet made changes."
            when 4 then "Puppet found errors."
        end
    end
end
```

4. When you want to test your changes on the client, run:

```
rake CLIENT=cookbook apply
```

Replace `cookbook` with the name of the client machine, or set the `CLIENT` environment variable so that Rake knows which machine to run Puppet on.

```
info: Caching catalog for cookbook
info: Applying configuration version '1292865016'
info: Creating state file /var/lib/puppet/state/state.yaml
notice: Finished catalog run in 0.03 seconds
```

5. If you want to see what changes Puppet would make, without actually changing anything, use the `--noop` flag:

```
task :noop => [:deploy] do
    client = ENV['CLIENT']
    sh "#{SSH} #{client} 'sudo puppet agent --test --noop'"
end
```

Now you can run:

```
rake noop
```

to get a preview of the changes.

How it works...

A Rakefile consists of a series of tasks, identified by the `task` keyword. The task definition is a set of steps, in this case the sequence of shell commands required to push your manifest changes to the master repo, and update the Puppetmaster's working copy.

Tasks can be linked, so that one depends on the other. For example, in our Rakefile the `apply` task is linked to `deploy`, so that whenever you run `rake apply`, Rake will make sure the `deploy` task is done first, then do the `apply` task.

There's more...

You can extend this Rakefile to automate more tasks, including running a syntax check on the Puppet manifests before updating them, and even bootstrapping a new machine with Puppet. Rake is a powerful tool and can be a big help in managing a large network with Puppet.

See also

- Using version control for your Puppet manifests
- Creating a distributed Puppet architecture based on Git
- Checking your manifests using commit hooks

Configuring Puppet's file server

Somewhere in every nuclear power plant control system, there's a file named `reactor.conf`. Deploying configuration files is one of the commonest uses of Puppet. Most non-trivial services need some kind of config file, and you can have Puppet push it to the client using a `file` resource like this:

```
file { "/opt/nginx/conf.d/app_production.conf":
    source => "puppet:///modules/app/app_production.conf",
}
```

The `source` parameter works like this: the first part after `puppet:///` is assumed to be the name of a **mount point**, and the remainder is treated as a path to the file.

```
puppet:///<mount point>/<path>
```

Most commonly the value of `<mount point>` is `modules`, as in the example above. In this case, Puppet will look for the file in:

```
manifests/modules/app/files/app_production.conf
```

`modules` is a mount point which Puppet treats specially: it expects the next path component to be the name of a module, and it will then look in the module's `files` directory for the remainder of the path.

However, Puppet lets you create custom mount points, which can have individual access control settings, and can be mapped to different locations on the Puppetmaster. In this recipe we'll see how to create and configure these custom mount points.

How to do it...

1. Add a stanza to the Puppetmaster's `fileserver.conf`, with the name of your mount point in square brackets, and the path where Puppet should look for data, like this:

```
[san]
    path /mnt/san/mydata/puppet
```

2. In your manifest, specify a file source using your mount point name, like this:

```
source => "puppet:///san/admin/users.htpasswd",
```

and Puppet will convert this to the path:

```
/mnt/san/mydata/puppet/admin/users.htpasswd
```

One good reason to create a custom mount point like this is to add some security. Let's say you have a top-secret password file which should only be deployed to the web server, and no other machine needs it. If someone can run Puppet on any machine that has a valid certificate to access the Puppetmaster, there's nothing to stop them executing a manifest like this:

```
file { "/home/cracker/goodstuff/passwords.txt":
    source => "puppet:///web/passwords.txt",
}
```

and they can retrieve the secret data. Indeed, anyone who can check out the Puppet repo or who has an account on the Puppetmaster could access this file. One way to avoid this is to put secret data into a special mount point with access control.

3. Add `allow` and `deny` parameters to your mount point definition in `fileserver.conf` like this:

```
[secret]
   /data/secret
   allow web.example.com
   deny *
```

How it works...

In this case, only `web.example.com` can access the file. The default is to deny all access, so the `deny *` line isn't strictly necessary, but it's good style to make it explicit. The web server can then use a `file` resource like this:

```
file { "/etc/passwords.txt":
   source => "puppet:///secret/passwords.txt",
}
```

If this manifest is executed on `web.example.com`, it will work, but on any other clients, it will fail.

There's more...

You can also specify an IP address instead of a hostname, optionally using **CIDR** (slash) notation or wildcards, like this:

```
allow 10.0.55.0/24
allow 192.168.0.*
```

See also

- Using modules
- Distributing directory trees
- Using multiple file sources

Running Puppet from cron

Is your Puppet sleeping on the job? By default, when you run the Puppet agent on a client, it will become a daemon (background process), waking up every 30 minutes to check for any manifest updates and apply them. If you want more control over when Puppet runs, you can trigger it using `cron` instead.

For example, if you have very many Puppet clients, you may want to deliberately stagger the Puppet run times to spread the load on the Puppetmaster. A simple way to do this is to set the minute or hour of the cron job time using a hash of the client hostname.

How to do it...

1. Use Puppet's `inline_template` function, which allows you to execute Ruby code, to :

   ```
   cron { "run-puppet":
       command => "/usr/sbin/puppet agent --test >/dev/null 2>&1",
       minute  => inline_template("<%= hostname.hash % 60 %>"),
   }
   ```

How it works...

Because each hostname produces a unique hash value, each client will run Puppet at a different minute past the hour. This hashing technique is useful for randomizing any cron jobs to improve the odds that they won't interfere with each other.

There's more...

Nothing for this section.

See also

- Randomly distributing cron job runs to reduce load
- Using embedded Ruby to supplement the Puppet language with 'inline_template'

Retrieving files from Puppet's filebucket

We all make mistakes; that's why pencils have erasers. Whenever Puppet changes a file on the client, it keeps a backup copy of the previous version. We can see this process in action if we make a change, however small, to an existing file:

```
root@cookbook:/etc/puppet# puppet agent --test
    info: Caching catalog for cookbook
    info: Applying configuration version '1293459139'
    --- /etc/sudoers    2010-12-27 07:12:20.421896753 -0700
    +++ /tmp/puppet-file20101227-1927-13hjvy6-0 2010-12-27
    07:13:21.645702932 -0700
    @@ -12,7 +12,7 @@

     # User alias specification
    -User_Alias SYSOPS = john
    +User_Alias SYSOPS = john,bob

    info: FileBucket adding /etc/sudoers as {md5}c07d0aa2d43d58ea7b5c5307
    f532a0b1
    info: /Stage[main]/Admin::Sudoers/File[/etc/sudoers]: Filebucketed /
    etc/sudoers to puppet with sum c07d0aa2d43d58ea7b5c5307f532a0b1
    notice: /Stage[main]/Admin::Sudoers/File[/etc/sudoers]/content:
    content changed '{md5}c07d0aa2d43d58ea7b5c5307f532a0b1' to '{md5}0d218
    c16bd31206e312c885884fa947d'
    notice: Finished catalog run in 0.45 seconds
```

The part we're interested in is this line:

```
    info: /Stage[main]/Admin::Sudoers/File[/etc/sudoers]: Filebucketed /
    etc/sudoers to puppet with sum c07d0aa2d43d58ea7b5c5307f532a0b1
```

Puppet creates an MD5 hash of the file's contents and uses this to create a **filebucket** path, based on the first few characters of the hash. The filebucket is where Puppet keeps backup copies of any files it replaces, and it's located by default in `/var/lib/puppet/clientbucket`:

- **root@cookbook:/etc/puppet# ls /var/lib/puppet/clientbucket/c/0/7/d/0/a/a/2/c07d0aa2d43d58ea7b5c5307f532a0b1**

```
contents  paths
```

You will see two files in the bucket location: `contents` and `paths`. The `contents` file contains, as you might expect, the original contents of the file. The `paths` file contains its original path.

It's easy to find the file if you know its content hash (as we did in this case). If you don't, it's helpful to create a table of contents of the whole filebucket by building an index file.

How to do it...

1. Create the index file using this command:

   ```
   find /var/lib/puppet/clientbucket -name paths -execdir cat {} \;
   -execdir pwd \; -execdir date -r {} +"%F %T" \; -exec echo \; >
   bucket.txt
   ```

2. Search the index file to find the file you're looking for:

   ```
   root@cookbook:/etc/puppet# cat bucket.txt
   /etc/sudoers
   /var/lib/puppet/clientbucket/c/0/7/d/0/a/a/2/c07d0aa2d43d58ea7b5c5
   307f532a0b1
   2010-12-27 07:13:21

   /etc/sudoers
   /var/lib/puppet/clientbucket/1/0/9/0/e/2/8/a/1090e28a70ebaae872c2e
   c78894f49eb
   2010-12-27 07:12:20
   ```

3. To retrieve the file once you know its bucket path, just copy the `contents` file to the original filename:

   ```
   cp /var/lib/puppet/clientbucket/1/0/9/0/e/2/8/a/1090e28a70ebaae872
   c2ec78894f49eb/contents /etc/sudoers
   ```

How it works...

The script will create a complete list of files in the filebucket, showing the original name of the file, the bucket path, and the modification date (in case you need to retrieve one of several previous versions of the file). Once you know the bucket path, then you can copy the file back into place.

There's more...

You can have Puppet create backup copies of the file in its original location, rather than in the filebucket. To do this, use the `backup` parameter in your manifest:

```
file { "/etc/sudoers":
    mode   => "440",
    source => "puppet:///modules/admin/sudoers",
    backup => ".bak",
}
```

Now if Puppet replaces the file, it will create a backup version in the same location with the extension `.bak`. To make this the default policy for all files, use:

```
File {
    backup => ".bak",
}
```

To disable backups altogether, use:

```
    backup => false,
```

Using autosign

In cryptography, as in life, you have to be careful what you sign. Normally, when you introduce a new client to the Puppetmaster, you need to generate a certificate request on the client, and then sign it on the master. However, you can skip this step by enabling **autosigning**.

How to do it...

1. Create the file `/etc/puppet/autosign.conf` on the Puppetmaster with the following contents:

   ```
   *.example.com
   ```

How it works...

Puppet checks any incoming certificate requests to see if they match a line in `autosign.conf`. Any certificate requests from clients with a hostname matching `*.example.com` will be automatically signed by the Puppetmaster.

Important: this is a potential security problem, since it amounts to trusting any client that can connect to the Puppetmaster. For this reason, autosigning is not recommended. If you do use it, make sure that the Puppetmaster is protected by a firewall which allows only approved clients or IP ranges to connect. A more secure approach is **pre-signing**.

There's more...

Nothing for this section.

See also

- Pre-signing certificates

Pre-signing certificates

Here's one I signed earlier. In general, if you want to automate adding a large number of clients, it's better to pre-generate the certificates on the Puppetmaster and then push them to the client as part of the build process. You can use `puppetca --generate <hostname>` to do this.

How to do it...

1. Generate a pre-signed certificate for `client1.example.com`:

   ```
   puppetca --generate client1.example.com
   ```

Puppet will now generate and sign a client certificate in the name of `client1.example.com`.

2. Transfer the required three files to the new client: the private key, the client certificate, and the CA certificate. These are found in the following locations:

   ```
   /etc/puppet/ssl/private_keys/client1.example.com.pem
   /etc/puppet/ssl/certs/client1.example.com.pem
   /etc/puppet/ssl/certs/ca.pem
   ```

Transfer these to the corresponding directories on the client, and it will then be authenticated without the certificate request step.

How it works...

No explanation required.

There's more...

Nothing for this section

See also

- Using `autosign` to quickly deploy server instances

Scaling Puppet using Passenger

Ever had to fire an employee because they were just getting old and slow? Puppet ships with a simple web server called Webrick to handle client connections to the Puppetmaster. This is fine for small numbers of servers, but you may find that as the number of clients increases (say 50-100+), the Puppetmaster becomes a performance bottleneck.

In order to scale Puppet to hundreds of servers, one approach is to switch to a high-performance web server such as Apache using the **Passenger** (`mod_rails`) extension. Puppet comes with the necessary configuration to run under Passenger, so all you need to do is install Apache and Passenger, and add a suitable virtual host. The following example uses Ubuntu 10.4; you can find instructions on the Puppet Labs website for how to do the same in Red Hat Linux, CentOS and other distributions at `http://projects.puppetlabs.com/projects/1/wiki/Using_Passenger`.

Getting ready

It will be helpful if you have available the source tarball for the version of Puppet you're running, because it provides several template files and configuration snippets which you can use to set up Passenger.

1. For example, if you're running Puppet 2.6.4, download this file:

   `http://puppetlabs.com/downloads/puppet/puppet-2.6.4.tar.gz`

If you are using a different version, you will find a suitable download link at `http://puppetlabs.com`.

2. Unpack the source tarball with:

   ```
   tar xzf puppet-2.6.4.tar.gz
   ```

How to do it...

1. Install Apache and Passenger, plus associated dependencies:

   ```
   apt-get install apache2 libapache2-mod-passenger rails librack-
   ruby libmysql-ruby
   gem install rack
   ```

2. Create the necessary directories for Passenger to find the Puppet configuration:

   ```
   /etc/puppet/rack
   /etc/puppet/rack/public
   ```

These directories should be owned by `root` and set mode 0755.

3. Create the `config.ru` file which will tell Passenger how to start the Puppet application. You can use the example file provided with the Puppet distribution:

   ```
   cp /tmp/puppet-2.6.4/ext/rack/files/config.ru /etc/puppet/rack/
   chown puppet /etc/puppet/rack/config.ru
   ```

For Puppet 2.6.4, it has the following contents:

```
# a config.ru, for use with every rack-compatible webserver.
# SSL needs to be handled outside this, though.

# if puppet is not in your RUBYLIB:
# $:.unshift('/opt/puppet/lib')

$0 = "master"

# if you want debugging:
# ARGV << "--debug"

ARGV << "--rack"
require 'puppet/application/master'
# we're usually running inside a Rack::Builder.new {} block,
# therefore we need to call run *here*.
run Puppet::Application[:master].run
```

4. You now need to create a virtual host for Apache to listen on the correct port and send requests to the Puppet application. Again, you can use the example provided with the Puppet distribution:

   ```
   cp /tmp/puppet-2.6.4/ext/rack/files/apache2.conf /etc/apache2/
   sites-available/puppetmasterd

   a2ensite puppetmasterd
   ```

The file contents will look something like this:

```
# you probably want to tune these settings
PassengerHighPerformance on
PassengerMaxPoolSize 12
PassengerPoolIdleTime 1500
# PassengerMaxRequests 1000
PassengerStatThrottleRate 120
```

```
RackAutoDetect Off
RailsAutoDetect Off

Listen 8140

<VirtualHost *:8140>
  SSLEngine on
  SSLProtocol -ALL +SSLv3 +TLSv1
  SSLCipherSuite ALL:!ADH:RC4+RSA:+HIGH:+MEDIUM:-LOW:-SSLv2:-EXP

  SSLCertificateFile      /etc/puppet/ssl/certs/cookbook.
bitfieldconsulting.com.pem
  SSLCertificateKeyFile   /etc/puppet/ssl/private_keys/cookbook.
bitfieldconsulting.com.pem
  SSLCertificateChainFile /etc/puppet/ssl/ca/ca_crt.pem
  SSLCACertificateFile    /etc/puppet/ssl/ca/ca_crt.pem
  # If Apache complains about invalid signatures on the CRL, you
can try disabling
  # CRL checking by commenting the next line, but this is not
recommended.
  SSLCARevocationFile     /etc/puppet/ssl/ca/ca_crl.pem
  SSLVerifyClient optional
  SSLVerifyDepth  1
  SSLOptions +StdEnvVars

  DocumentRoot /etc/puppet/rack/public/
  RackBaseURI /
  <Directory /etc/puppet/rack/>
    Options None
    AllowOverride None
    Order allow,deny
    allow from all
  </Directory>
</VirtualHost>
```

5. Edit this file to set the values of `SSLCertificateFile` and `SSLCertificateKeyFile` to your own certificates (it's easiest if you've already run Puppet at least once, to create these certificates).
6. You will also need to enable Passenger and `mod_ssl` in Apache:

   ```
   a2enmod passenger ssl
   ```

7. Add the following lines to your `/etc/puppet/puppet.conf`:

   ```
   ssl_client_header = SSL_CLIENT_S_DN
   ssl_client_verify_header = SSL_CLIENT_VERIFY
   ```

8. Stop your existing Puppetmaster if it is running.
9. Start Apache:

```
/etc/init.d/apache2 restart
```

10. If everything has worked, you will be able to run Puppet as usual:

```
# puppet agent --test
    info: Caching catalog for cookbook.bitfieldconsulting.com
    info: Applying configuration version '1294145142'
    notice: Finished catalog run in 0.25 seconds
```

How it works...

Instead of using Puppet's built-in webserver, which is rather slow and can only handle one connection at once, you're using the high-performance multi-threaded Apache webserver. Puppet is embedded as an application using the Rack framework, which is much more efficient. You should find that you can handle many more clients and more frequent Puppet runs using the Apache + Passenger configuration, and that the impact on server memory and performance is less than using the standard Puppetmaster daemon.

There's more...

Here is an example Puppet manifest which will implement the above steps for you (on an Ubuntu system):

```
class puppet::passenger {
    package { [ "apache2-mpm-worker",
                "libapache2-mod-passenger",
                "librack-ruby",
                "libmysql-ruby" ]:
        ensure => installed,
    }

    service { "apache2":
        enable  => true,
        ensure  => running,
        require => Package["apache2-mpm-worker"],
    }

    package { "rack":
        provider => gem,
        ensure   => installed,
    }
```

```
        file { [ "/etc/puppet/rack",
                 "/etc/puppet/rack/public" ]:
            ensure => directory,
            mode   => "755",
        }

        file { "/etc/puppet/rack/config.ru":
            source => "puppet:///modules/puppet/config.ru",
            owner  => "puppet",
        }

        file { "/etc/apache2/sites-available/puppetmasterd":
            source => "puppet:///modules/puppet/puppetmasterd.conf",
        }

        file { "/etc/apache2/sites-enabled/puppetmasterd":
            ensure => symlink,
            target => "/etc/apache2/sites-available/puppetmasterd",
        }

        exec { "/usr/sbin/a2enmod ssl":
            creates => "/etc/apache2/mods-enabled/ssl.load",
        }
    }
```

For more details, or if you run into problems, consult the Puppet-on-Passenger documentation: `http://projects.puppetlabs.com/projects/1/wiki/Using_Passenger`

See also

- Creating a decentralized Puppet architecture

Creating a decentralized Puppet architecture

Some systems - like the Mafia - run best when they're decentralized. The most common way to use Puppet is to run a Puppetmaster server, which Puppet clients can then connect to and receive their manifests. However, you can run Puppet directly on a manifest file to have it executed (you'll normally want to use the `-v` switch to enable verbose mode, so you can see what's happening):

```
# puppet -v manifest.pp
    info: Applying configuration version '1294313350'
```

Or even supply a manifest directly on the command line:

```
# puppet -e "file { '/tmp/test': ensure => present }"
   notice: /Stage[main]//File[/tmp/test]/ensure: created
```

In other words, if you can arrange to distribute a suitable manifest file to a client machine, you can have Puppet execute it directly without the need for a central Puppetmaster. This removes the performance bottleneck of a single master server, and also eliminates a single point of failure. It also avoids having to sign and exchange SSL certificates when provisioning a new client machine.

There are many ways you could deliver the manifest file to the client, but Git (or another version control system such as Mercurial or Subversion) does most of the work for you. You can edit your manifests in a local working copy, commit them to Git and push them to a central repo, and from there they can be automatically distributed to the client machines.

Getting ready

If your Puppet manifests aren't already in Git, follow the steps in:

- Using version control for your Puppet manifests

How to do it...

1. Make a bare clone of your Puppet repo on the client:

   ```
   # git clone --bare ssh://git@repo.example.com/var/git/puppet
   ```

2. Copy the contents of this repo into your `/etc/puppet/` directory using:

   ```
   # git archive --format=tar HEAD | (cd /etc/puppet && tar xf -)
   ```

3. Run Puppet on your `site.pp` file:

   ```
   # puppet -v /etc/puppet/manifests/site.pp
   info: Applying configuration version '1294313353'
   ```

4. Once this is working, the next step is to have the configuration repo automatically push out changes to the clients. With Git, you can do this using **remotes**, like so:

   ```
   # git remote add web ssh://git@web1.example.com/etc/puppet
   ```

If you have multiple client machines, you can add more URLs to the same remote:

```
# git remote set-url --add webs ssh://git@web2.example.com/etc/puppet
# git remote set-url --add webs ssh://git@web3.example.com/etc/puppet
...
```

or simply edit the Git config file (`.git/config`) like this:

```
[remote "web"]
    url = ssh://git@web1.example.com/etc/puppet
    url = ssh://git@web2.example.com/etc/puppet
    url = ssh://git@web3.example.com/etc/puppet
    ...
```

5. Now you can push to any client machine, or group of machines, from the repo server:

   ```
   # git push web
   ```

6. The final step is to have the client machine update its `/etc/puppet` directory whenever it receives a push from the repo server. You can do this using a Git post-receive hook. In your bare repo, create the file `hooks/post-receive` and make it executable (mode 0755):

   ```
   #!/bin/sh
   git archive --format=tar HEAD | (cd /etc/puppet && tar xf -)
   ```

How it works...

Instead of contacting the Puppetmaster to receive their compiled manifest, each client compiles its own from a local copy of the manifest source. This is updated every time you push updates from the Git server (or from your working checkout). This is more efficient in network bandwidth, as clients don't have to contact the Puppetmaster on every run. It also eliminates a single point of failure, as clients can be updated from anywhere.

Using a decentralized Puppet architecture based on Git as outlined here gives you a great deal of flexibility. You can configure access controls and permissions using SSH keys, and allow each client machine or group only as much access as it needs. Manifests for a database server group, for example, can be made available only to those machines which need it.

While it requires some extra work to set up, and is not necessary for most small organisations, this way of deploying Puppet gives you extra flexibility and control for the most demanding environments.

There's more...

If you want to have Puppet apply the changes every time they are pushed, you can edit the `post-receive` script to do this, or take any other action you want. Alternatively, you could run Puppet manually, or from `cron` as described earlier in this chapter - just remember to run `puppet` rather than `puppet agent`.

You can find a more detailed discussion of this architecture in Stephen Nelson-Smith's article: `http://bitfieldconsulting.com/scaling-puppet-with-distributed-version-control`

See also

- Scaling Puppet using Passenger
- Using version control

2
Monitoring, Reporting, and Troubleshooting

"Found problem more than one. However, this does not mean that relevant part is thing by mistake. Could be fertilized by special purpose in other application program."

— Error message

In this chapter, we will cover:

- Generating reports
- Emailing log messages containing specific tags
- Creating graphical reports
- Producing automatic HTML documentation
- Drawing dependency graphs
- Testing your Puppet manifests
- Doing a dry run
- Detecting compilation errors
- Using commit hooks
- Understanding Puppet errors
- Logging command output
- Logging debug messages
- Inspecting configuration settings
- Using tags
- Using run stages
- Using environments

Introduction

We've all had the experience of sitting in an exciting presentation about some new technology, and rushing home to play with it. Of course, once you start experimenting with it, you immediately run into problems. What's going wrong? Why doesn't it work? How can I see what's happening under the hood? This chapter will help you answer some of these questions, and give you the tools to solve common Puppet problems. We'll also see how to generate useful reports on your Puppet infrastructure, and how Puppet can help you monitor and troubleshoot your network as a whole.

Generating reports

"What the world really needs is more love and less paperwork."

— Pearl Bailey

Truth is the first casualty of large infrastructures. If you're managing a lot of machines, Puppet's reporting facility can give you some valuable information on what's actually happening out there.

How to do it...

1. To enable reports, just add this to a client's `puppet.conf`:

   ```
   report = true
   ```

How it works...

With reporting enabled, Puppet will generate a report file on the Puppetmaster, containing data such as:

- Time to fetch config from the Puppetmaster
- Total time of the run
- Log messages output during the run
- List of all resources in the client's manifest
- Whether Puppet changed each resource
- Whether a resource was out of sync with the manifest

By default, these reports are stored in `/var/lib/puppet/reports`, but you can specify a different destination using the `reportdir` option. You can either create your own scripts to process these reports (which are in the standard **YAML** format), or use a tool such as Puppet Dashboard to get a graphical overview of your network.

There's more...

Here are a couple more tips for getting the best from Puppet's reports.

Enabling reports on the command line

If you just want one report, or you don't want to enable reporting for all clients, you can add the `--report` switch to the command line when you run Puppet manually:

```
# puppet agent --test --report
```

You can also see some statistics about a Puppet run by supplying the `--summarize` switch:

```
# puppet agent --test --summarize
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1306169315'
notice: Finished catalog run in 0.58 seconds
Changes:
Events:
Resources:
            Total: 7
Time:
   Config retrieval: 3.65
       Filebucket: 0.00
         Schedule: 0.00
```

Logging Puppet messages to syslog

Puppet can also send its log messages to the Puppetmaster's syslog, so that you can analyze them with standard syslog tools. To enable this, set the following option in the Puppetmaster's `puppet.conf`:

```
[master]
reports = store,log
```

`store` is the default report type (which writes the reports to `/var/lib/puppet/reports`), and `log` tells Puppet to also send messages to the syslog.

See also

- Creating graphical reports using RRD
- Using 'notice' and 'notify' to print debug information
- Seeing an overview of your network with Dashboard

Emailing log messages containing specific tags

If, like most sysadmins, you don't get enough email, you'll be looking for a way to generate more. Another type of Puppet report is called `tagmail`. This will email the log messages to any address you specify.

How to do it...

1. Add `tagmail` to the comma-separated list of reports in `puppet.conf`:

   ```
   [master]
   reports = store,tagmail
   ```

2. Add some **tags** and associated e-mail addresses in the file `/etc/puppet/tagmail.conf`. For example, this line will e-mail all log messages to me:

   ```
   all: john@bitfieldconsulting.com
   ```

3. When Puppet runs, you will get an email that looks like this:

   ```
   From: report@cookbook.bitfieldconsulting.com
   Subject: Puppet Report for cookbook.bitfieldconsulting.com
   To: john@bitfieldconsulting.com

   Mon Jan 17 08:42:30 -0700 2011 //cookbook.bitfieldconsulting.com/
   Puppet (info): Caching catalog for cookbook.bitfieldconsulting.com
   Mon Jan 17 08:42:30 -0700 2011 //cookbook.bitfieldconsulting.com/
   Puppet (info): Applying configuration version '1295278949'
   ```

How it works...

Puppet looks at each line in `tagmail.conf` and sends any messages matching the tag to the email address specified. The special tag `all` matches all messages. The tag `err` matches errors:

```
err: john@bitfieldconsulting.com
```

You can list as many rules as you like in the `tagmail.conf` file, and Puppet will send emails for all rules that match. In this example, errors go to one address, and webserver-related messages go to another:

```
err: puppetmaster@example.com
webserver: webteam@example.com
```

There's more…

The `tagmail` reports are a powerful feature which you may need to experiment with a bit to get the most out of them. Here are some tips.

What are tags?

Tags are more fully explained later in this book, but for reporting purposes, it's enough to know that a tag can be the name of a node or a class (so the tag `webserver` is matched if a machine includes the class `webserver`), or you can add a tag explicitly using the `tag` function like this:

```
class exim {
     tag("email")
     service { "exim4":
         ensure => running,
         enable => true,
     }
 }
```

Specifying multiple tags, or excluding tags

You can specify a list of comma-separated tags in `tagmail.conf`, and also exclude certain tags by using an exclamation point (`!`):

```
all, !webserver: puppetmaster@example.com
```

Sending reports to multiple email addresses

You can have rules send messages to multiple, comma-separated email addresses:

```
err: puppetmaster@example.com, sysadmin@example.com
```

See also

- Generating reports
- Creating graphical reports
- Using tags and `defined`

Creating graphical reports

Let's face it, bosses like pretty pictures. Puppet can produce report data in a form suitable for processing by the **RRD** graph library, to produce a graphical representation of the metrics such as the run time on each client.

Getting ready

You will need to install the RRD tools and libraries for Ruby on your system. For Ubuntu, run:

```
# apt-get install rrdtool librrd-ruby
```

How to do it...

5.Add the `rrdgraph` report type to your `puppet.conf`:

```
reports = store,rrdgraph
```

How it works...

For each run, Puppet will record data in the client's RRD directory (by default `/var/lib/puppet/rrd/<clientname>`). It will create PNG-format graphs of events, resources, and retrieval time, while the raw data is available to you in the `.rrd` files if you want to process it further using third-party RRD tools.

There's more...

For more detailed reporting and graphing, you can use Puppet Dashboard.

See also

- Using Dashboard

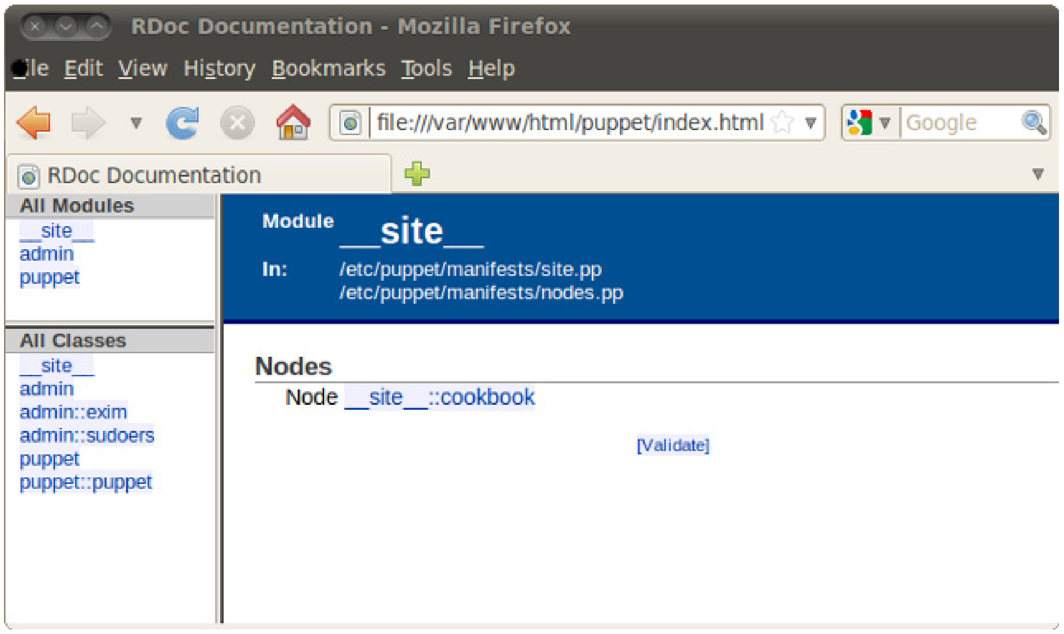
Producing automatic HTML documentation

Like most engineers, I never read the manual, unless and until the product actually catches fire. However, as your manifests get bigger and more complex, it can be helpful to create HTML documentation for your nodes and classes using Puppet's automatic documentation tool, `puppet doc`.

How to do it...

1. Run `puppet doc` over your manifest:

```
 puppet doc --all --outputdir=/var/www/html/puppet --mode rdoc --
manifestdir=/etc/puppet/manifests/
```


RDoc Documentation - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///var/www/html/puppet/index.html
Google
RDoc Documentation

All Modules
__site__
admin
puppet

All Classes
__site__
admin
admin::exim
admin::sudoers
puppet
puppet::puppet

Module __site__
In: /etc/puppet/manifests/site.pp
/etc/puppet/manifests/nodes.pp

Nodes
Node __site__::cookbook

[Validate]

How it works...

`puppet doc` creates a structured HTML documentation tree in `/var/www/html/puppet` similar to that produced by **RDoc**, the popular Ruby documentation generator. This makes it easier to understand how different parts of the manifest relate to one another, as you can click on an included class name and see its definition, for example.

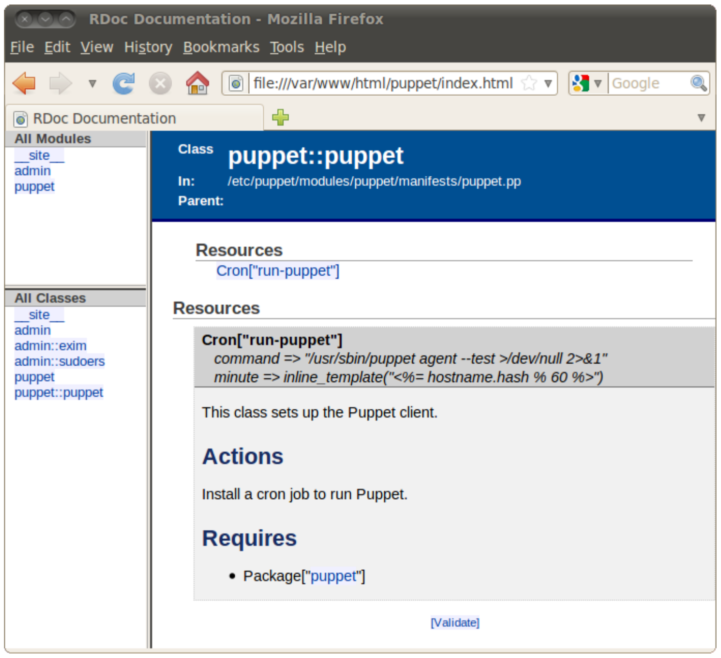
There's more...

`puppet doc` will generate basic documentation of your manifests as they stand, but you can include more useful information by adding comments to your manifest files, using the standard RDoc syntax. Here's an example of some documentation comments added to a class:

```
class puppet {
   # This class sets up the Puppet client.
   #
```

```
        # ==Actions
        # Install a cron job to run Puppet.
        #
        # ==Requires
        # * Package["puppet"]
        #
        cron { "run-puppet":
            command => "/usr/sbin/puppet agent --test >/dev/null 2>&1",
            minute  => inline_template("<%= hostname.hash.abs % 60 %>"),
        }
    }
```

Your comments are added to the documentation for each class in the resulting HTML files, like this:



Class **puppet::puppet**
In: /etc/puppet/modules/puppet/manifests/puppet.pp
Parent:

Resources
Cron["run-puppet"]

Resources

Cron["run-puppet"]
command => "/usr/sbin/puppet agent --test >/dev/null 2>&1"
minute => inline_template("<%= hostname.hash % 60 %>")

This class sets up the Puppet client.

Actions

Install a cron job to run Puppet.

Requires

- Package["puppet"]

Drawing dependency graphs

Dependencies can get complicated quickly, and it's easy to end up with a **circular dependency** (where A depends on B which depends on A) which will cause Puppet to complain and stop work. Fortunately, Puppet's `--graph` option makes it easy to generate a diagram of your resources and the dependencies between them, which can be a big help in fixing such problems.

Getting ready...

1. nstall the `graphviz` package for viewing the diagram files:

   ```
   # apt-get install graphviz
   ```

How to do it...

1. Create the file `/etc/puppet/modules/admin/manifests/ntp.pp` with the following code containing a circular dependency:

   ```
   class admin::ntp {
       package { "ntp":
           ensure => installed,
           **require => File["/etc/ntp.conf"],**
       }

       service { "ntp":
           ensure  => running,
           require => Package["ntp"],
       }

       file { "/etc/ntp.conf":
           source  => "puppet:///modules/admin/ntp.conf",
           notify  => Service["ntp"],
           require => Package["ntp"],
       }
   }
   ```

2. Copy your existing `ntp.conf` file into Puppet:

   ```
   # cp /etc/ntp.conf /etc/puppet/modules/admin/files
   ```

3. Include this class on a node:

   ```
   node cookbook {
         include admin::ntp
   }
   ```

4. Run Puppet:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   err: Could not apply complete catalog: Found dependency cycles in
   the following relationships: File[/etc/ntp.conf] => Package[ntp],
   Package[ntp] => File[/etc/ntp.conf], Package[ntp] => Service[ntp],
   File[/etc/ntp.conf] => Service[ntp]; try using the '--graph'
   ```

```
option and open the '.dot' files in OmniGraffle or GraphViz
notice: Finished catalog run in 0.42 seconds
```

5. Check the graph files have been created:

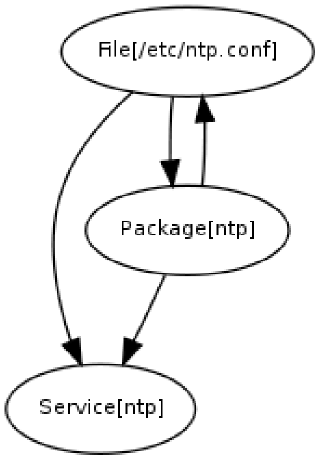
```
 # ls /var/lib/puppet/state/graphs/
 expanded_relationships.dot  relationships.dot  resources.dot
```

13.Create a graphic of the relationships graph:

```
 # dot -Tpng -o relationships.png /var/lib/puppet/state/graphs/
relationships.dot
```

14.View the graphic:

```
 # eog relationships.png
```



How it works...

When you run `puppet --graph` (or enable the `graph` option in `puppet.conf`) Puppet will generate three graphs in DOT format (a graphics language):

- `resources.dot` - showing the hierarchical structure of your classes and resources, but without dependencies
- `relationships.dot` - showing the dependencies between resources as arrows, as above
- `expanded_relationships.dot` - a more detailed version of the relationships graph

The `dot` tool (part of the `graphviz` package) will convert these to an image format such as PNG for viewing.

In the relationships graph, each resource in your manifest is shown as a balloon, with arrowed lines connecting them to indicate the dependencies. You can see that in our example, the dependencies between `File["/etc/ntp.conf"]` and `Package["ntp"]` form a circle.

To fix the circular dependency problem, all you need to do is remove one of the dependency lines and so break the circle.

There's more...

Resource and relationship graphs can be useful even when you don't have a bug to find. If you have a very complex network of classes and resources, for example, studying the resources graph can help you see where to simplify things. Similarly, when dependencies become too complicated to understand from reading the manifest, the graphs can be a much more useful form of documentation.

Testing your Puppet manifests

"If all else fails, immortality can always be assured by spectacular error."

— J.K. Galbraith

Trouble has a way of sneaking up on you like a windshield on a bug. The standard checks provided by monitoring tools like Nagios don't always cover everything you want to monitor. While metrics such as load average and disk space can be useful problem indicators, I like to be able to get higher-level information about the applications and services my machines provide.

For example, if you are running a web application, it's not enough to know that the web server is listening to connections on port 80 and responding with an HTTP 200 OK status. It could just be returning the default Apache welcome page.

If your web application is an online store, for example, you might want to check the following:

- Do we see expected text in the returned page (for example, "Welcome to FooStore")?
- Can we log in as a user (if the application supports sessions)?
- Can we search for a product and see an expected result?
- Is the response time satisfactory?

This kind of monitoring - focusing on the **behaviour** of the application, rather than operational metrics about the server itself - is sometimes called **behaviour-driven monitoring**.

Just as developers often use behaviour-driven tests to verify that the application does what it should when they make code changes, you can use behaviour-driven monitoring to monitor it continuously in production.

In fact, thanks to a tool called `cucumber-nagios`, you can run the same tests the developers use. Lindsay Holmwood's wrapper for the popular **Cucumber** testing framework lets you run Cucumber-based tests under Nagios as though they were standard Nagios metrics.

Getting ready

1. To install `cucumber-nagios`, you will need a few dependencies first. If you are on Ubuntu or Debian, you will probably need to install RubyGems from source, as `cucumber-nagios` needs RubyGems 1.3.6 or higher. Download the tarball from the RubyGems site: `http://rubygems.org/pages/download`
2. Unpack it and run `ruby setup.rb` to build and install the package.
3. Next, you need to install a couple more dependencies:

   ```
   # apt-get install ruby1.8-dev libxml2-dev
   ```

4. Finally, you can install `cucumber-nagios` itself:

   ```
   # gem install cucumber-nagios
   ```

How to do it...

1. Once the gem and all its dependencies have been installed, you can start writing Cucumber tests. To do this, first use `cucumber-nagios` to help create a project directory with everything you will need:

   ```
   # cucumber-nagios-gen project mytest
   Generating with project generator:
        [ADDED]  features/steps
        [ADDED]  features/support
        [ADDED]  .gitignore
        [ADDED]  .bzrignore
        [ADDED]  lib/generators/feature/%feature_name%.feature
        [ADDED]  Gemfile
        [ADDED]  bin/cucumber-nagios
        [ADDED]  lib/generators/feature/%feature_name%_steps.rb
        [ADDED]  README

   Your new cucumber-nagios project can be found in /root/mytest.
   ```

```
 Next, install the necessary RubyGems with:

     bundle install

 Your project has been initialised as a git repository.
```

2. It's a good idea to run `bundle install` inside the project directory, as `cucumber-nagios` advises you. This will bundle all the dependencies for `cucumber-nagios` inside the directory. Then you can move the project directory to any machine and it will work.

```
 # cd mytest
 # bundle install
```

3. Now we can start writing a test. As an example, let's test the home page on Google:

```
 # cucumber-nagios-gen feature www.google.com home
 Generating with feature generator:
 [ADDED] features/www.google.com/home.feature
 [ADDED] features/www.google.com/steps/home_steps.rb
```

4. If you edit the `home.feature` file, you will find that `cucumber-nagios` has generated a basic first test for you:

```
 Feature: www.google.com
   It should be up

   Scenario: Visiting home page
     When I go to "http://www.google.com"
     Then the request should succeed
```

5. You can run this from the project directory with:

```
 # cucumber --require features features/www.google.com/home.
feature
 Feature: www.google.com
   It should be up


   Scenario: Visiting home page           # features/www.google.
com/home.feature:4
     When I go to "http://www.google.com" # features/steps/http_
steps.rb:11
     Then the request should succeed      # features/steps/http_
steps.rb:64
```

```
 1 scenario (1 passed)
 2 steps (2 passed)
 0m0.176s
```

6. Assuming this works (if it doesn't, call Google), all you need to do to make this feature a Nagios check is to run it with `cucumber-nagios` instead of `cucumber`:

```
 # bin/cucumber-nagios features/www.google.com/home.feature
 CUCUMBER OK - Critical: 0, Warning: 0, 2 okay | passed=2;
failed=0; nosteps=0; total=2; time=0
```

How it works...

Any script can be a Nagios monitoring plugin: it just has to return the appropriate exit status (`0` for OK, `1` for warning, `2` for critical). `cucumber-nagios` wraps Cucumber tests to do this, and also print out useful information which Nagios will report via the alert or the web interface.

There's more...

By itself, this doesn't do anything very useful. However, Cucumber lets you write quite sophisticated interaction scripts with websites: you can fill in form fields, search, click buttons, match text on the page, and so on. Whatever features of your web application or service you want to monitor, figure out first what a user would do in a web browser, then automate those steps with Cucumber to create the monitoring script.

You can find out more about how to write tests for `cucumber-nagios` on the Cucumber website: `http://cukes.info/`

Doing a dry run

"No alarms and no surprises."

— Radiohead

I hate surprises. Sometimes your Puppet manifest doesn't do exactly what you expected, or perhaps someone else has checked in changes you didn't know about. Either way, it's good to know exactly what Puppet is going to do before it does it.

If it would update a config file and restart a production service, for example, this could result in unplanned downtime. Also, sometimes manual configuration changes are made on a server which Puppet would overwrite.

To avoid these problems, you can use Puppet's **dry run** mode (also called `noop` mode, for 'no operation').

How to do it...

1. Run Puppet with the `--noop` switch:

```
 # puppet agent --test --noop
 info: Connecting to sqlite3 database: /var/lib/puppet/state/
clientconfigs.sqlite3
 info: Caching catalog for cookbook.bitfieldconsulting.com
 info: Applying configuration version '1296492323'
 --- /etc/exim4/exim4.conf   2011-01-17 08:13:34.349716342 -0700
 +++ /tmp/puppet-file20110131-20189-127zyug-0    2011-01-31
09:45:27.792843709 -0700
 @@ -1,4 +1,5 @@
  #########
 +# allow spammers to use our host as a relay
  #########
 notice: /Stage[main]/Admin::Exim/File[/etc/exim4/exim4.conf]/
content: is {md5}02798714adc9c7bf82bf18892199971a, should be {md5}
6f46256716c0937f3b6ffd6776ed059b (noop)
 info: /Stage[main]/Admin::Exim/File[/etc/exim4/exim4.conf]:
Scheduling refresh of Service[exim4]
 notice: /Stage[main]/Admin::Exim/Service[exim4]: Would have
triggered 'refresh' from 1 events
 notice: Finished catalog run in 0.90 seconds
```

How it works...

In `noop` mode, Puppet does everything it would normally, with the exception of actually making any changes to the machine. It tells you what it would have done, and you can compare this with what you expected to happen. If there are any differences, double-check the manifest or the current state of the machine.

Note that Puppet warns us it would have restarted the `exim` service, due to a config file update. This may or may not be what we want, but it's useful to know in advance. I make it a rule, when applying any non-trivial changes on production servers, to run Puppet in `noop` mode first, and verify what's going to happen.

There's more...

You can also use dry run mode as a simple auditing tool. It will tell you if any changes have been made to the machine since Puppet last applied its manifest. Some organisations require all config changes to be made with Puppet, which is one way of implementing a change control process. Unauthorized changes can be detected using Puppet in dry run mode and you can then decide whether to merge the changes back into the Puppet manifest, or undo them.

See also

- Auditing resources

Detecting compilation errors

Normally, when running in daemon mode, Puppet will ignore any compilation errors in the manifest and just apply the last known working version from its cache. This behaviour is governed by the `usecacheonfailure` config setting, and it defaults to `true`:

```
# puppet --genconfig |grep usecacheonfailure
  # usecacheonfailure = true
```

It's worth noting that when you apply manifests by hand using `puppet agent --test`, this doesn't happen: Puppet will complain and refuse to do anything if there is an error in the manifest. That's because the `--test` switch is shorthand for the following options:

```
# puppet agent --onetime --verbose --ignorecache --no-daemonize --no-
usecacheonfailure
```

Because `usecacheonfailure` is on when Puppet runs as a daemon, sometimes you won't notice mistakes in a manifest for a while, as Puppet keeps on silently running an old version of the manifest instead of complaining.

How to do it...

1. If you want to change this behaviour, set the following value in `puppet.conf`:

```
usecacheonfailure = false
```

How it works...

With this option set, Puppet will immediately complain about errors and refuse to run until they are corrected.

There's more...

Nothing for this section.

Understanding Puppet errors

Puppet's error messages can be confusing, and sometimes don't contain much helpful information about how to actually resolve the problem.

Often the first step is simply to search the web for the error message text and see what explanations you can find for the error, along with any helpful advice about fixing it. Here are some of the most common puzzling errors, with possible explanations:

Could not evaluate: Could not retrieve information from source(s)

You specified a `source` parameter for a file and Puppet couldn't find this source. Check the file is present and has been checked in, and also that the source path is correct.

change from absent to file failed: Could not set 'file on ensure: No such file or directory

This is often caused by Puppet trying to write a file to a directory that doesn't exist. Check that the directory either exists already or is defined in Puppet, and that the file resource requires the directory (so that the directory is always created first).

undefined method `closed?' for nil:NilClass

This unhelpful error message is roughly translated as "something went wrong". It tends to be a catch-all error caused by many different problems, but you may be able to determine what is wrong from the name of the resource, the class, or the module. One trick is to add the `--debug` switch, to get more useful information:

```
# puppet agent --test --debug
```

If you check your Git history to see what was touched in the most recent change, this may be another way to identify what's upsetting Puppet.

Could not parse for environment --- "--- production": S yntax error at end of file at line 1

This can be caused by mistyping command line options: for example, if you type `puppet -verbose` instead of `puppet --verbose`. That kind of error can be hard to see.

Could not request certificate: Retrieved certificate does not match private key; please remove certificate from server and regenerate it with the current key

Either the node's SSL host key has changed, or Puppet's SSL directory has been deleted, or you are trying to request a certificate for a machine with the same name as an existing node. Generally the simplest way to fix this is to remove Puppet's SSL directory on the client (usually this is `/etc/puppet/ssl`) and run `puppet cert --clean <nodename>` on the Puppetmaster. Then run Puppet again, and it should generate a certificate request correctly.

Duplicate definition: X is already defined in [file] at line Y; cannot redefine at [file] line Y

This one has caused me a bit of puzzlement in the past. Puppet's complaining about a duplicate definition, and normally if you have two resources with the same name, Puppet will helpfully tell you where they are both defined. But in this case, it's indicating the same file and line number for both. How can one resource be a duplicate of itself?

The answer is if it's a `define`. If you create two instances of a `define`, you'll also have two instances of all the resources contained within the `define`, and they need to have distinct names. For example:

```
        define check_process() {
            exec { "is-process-running?":
                command => "/bin/ps ax |/bin/grep ${name} >/tmp/
   pslist.${name}.txt",
            }
        }

        check_process { "exim": }
        check_process { "nagios": }


    # puppet agent --test
    info: Retrieving plugin
    err: Could not retrieve catalog from remote server: Error 400 on
```

```
ERVER: Duplicate definition: Exec[is-process-running?] is already
defined in file /etc/puppet/manifests/nodes.pp at line 22; cannot
redefine at /etc/puppet/manifests/nodes.pp:22 on node cookbook.
bitfieldconsulting.com
 warning: Not using cache on failed catalog
 err: Could not retrieve catalog; skipping run
```

Because the `exec` resource is named `is-process-running?` and this is the same whatever you pass to the `define`, Puppet will refuse to create two instances of it. The solution is to include the name of the instance in the title of each resource:

```
 exec { "is-process-${name}-running?":
             command => "/bin/ps ax |/bin/grep ${name} >/tmp/
pslist.${name}.txt",
         }
```

Logging command output

When you use `exec` resources to run commands on the node, it's not always easy to find out why they haven't worked. Puppet will give you an error message like this if a command returns a non-zero exit status:

```
 err: /Stage[main]//Node[cookbook]/Exec[this-will-fail]/returns:
change from notrun to 0 failed: /bin/ls file-that-doesnt-exist
returned 2 instead of one of [0] at /etc/puppet/manifests/nodes.pp:10
```

Often we would like to see the actual output from the command that failed, rather than just the numerical exit status. You can do this with the `logoutput` parameter.

How to do it...

1. Define an `exec` resource with the `logoutput` parameter like this:

   ```
   exec { "this-will-fail":
       command   => "/bin/ls file-that-doesnt-exist",
       logoutput => on_failure,
   }
   ```

How it works...

Now if the command fails, Puppet will also print its output:

```
 notice: /Stage[main]//Node[cookbook]/Exec[this-will-fail]/returns: /bin/
ls: cannot access file-that-doesnt-exist: No such file or directory
 err: /Stage[main]//Node[cookbook]/Exec[this-will-fail]/returns: change
from notrun to 0 failed: /bin/ls file-that-doesnt-exist returned 2
```

```
instead of one of [0] at /etc/puppet/manifests/nodes.pp:11
```

There's more...

You can set this to be the default for all exec resources by defining:

```
Exec {
    logoutput => on_failure,
}
```

If you always want to see the command output, whether it succeeds or fails, use:

```
logoutput => true,
```

Logging debug messages

It can be very helpful when debugging problems if you can print out information at a certain point in the manifest. This is a good way to tell, for example, if a variable isn't defined or has an unexpected value. Sometimes it's useful just to know that a particular piece of code has been run. Puppet's `notify` resource lets you print out such messages.

How to do it...

1. Define a `notify` resource in your manifest at the point you want to investigate:

```
notify { "Got this far!": }
```

How it works...

When this resource is compiled Puppet will print out the message:

```
notice: Got this far!
```

There's more...

If you're the kind of brave soul who likes experimenting, and I hope you are, you'll probably find yourself using debug messages a lot to figure out why your code doesn't work. So knowing how to get the most out of Puppet's debugging features can be a great help.

Printing out variable values

You can reference variables in the message:

```
notify { "operatingsystem is $operatingsystem": }
```

And Puppet will interpolate the values in the printout:

```
notice: operatingsystem is Ubuntu
```

Printing the full resource path

For more advanced debugging, you may want to use the `withpath` parameter to see in which class the `notify` was executed:

```
notify { "operatingsystem is $operatingsystem":
    withpath => true,
}
```

Now the notify message will be prefixed with the complete resource path:

```
notice: /Stage[main]/Nagios::Target/Notify[operatingsystem is Ubuntu]/
message: operatingsystem is Ubuntu
```

Logging messages on the Puppetmaster

Sometimes you just want to log a message on the Puppetmaster, without generating extra output on the client. You can use the `notice` function to do this:

```
notice("I am running on node $fqdn")
```

Now when you run Puppet, you will not see any output on the client, but on the Puppetmaster a message like this will be sent to the syslog:

```
Jan 31 11:51:38 cookbook puppet-master[22640]: (Scope(Node[cookbook])) I
am running on node cookbook.bitfieldconsulting.com
```

Inspecting configuration settings

You already know that Puppet's configuration settings are stored in `puppet.conf`, but any parameter not mentioned in that file will take a default value. How can you see the value of any configuration parameter, regardless of whether or not it's explicitly set in `puppet.conf`? The answer is to use Puppet's `--genconfig` switch.

How to do it...

1. Run

   ```
   # puppet --genconfig
   ```

How it works...

This will output every configuration parameter and its value (and there are lots of them). It does, however, include helpful comments explaining what each parameter does.

To find the specific value you're interested in, you can use grep like this:

```
# puppet --genconfig |grep "reportdir ="
    reportdir = /var/lib/puppet/reports
```

Using tags

Sometimes one Puppet class needs to know about another - or, at least, to know whether or not it's present. For example, a class that manages the firewall may need to know whether the node is a web server.

Puppet's `tagged` function will tell you whether a named class or resource is present in the manifest for this node. You can also apply arbitrary tags to a node or class and check for the presence of these tags.

1. To help you find out if you're running on a particular node or class of node, all nodes are automatically tagged with the node name and the names of any parent nodes it inherits from.

   ```
   node bitfield_server {
       include bitfield
   }

   node cookbook inherits bitfield_server {
       if tagged("cookbook") {
           notify { "this will succeed": }
       }
       if tagged("bitfield_server") {
           notify { "so will this": }
       }
   }
   ```

2. So that you can tell whether a particular class is included on this node, all nodes are automatically tagged with the names of all the classes they include, and their parent classes.

   ```
   include apache::port8000

   if tagged("apache::port8000") {
       notify { "this will succeed": }
   }
   ```

```
if tagged("apache") {
    notify { "so will this": }
}
```

3. If you want to set an arbitrary tag on a node, use the `tag` function:

```
tag("old-slow-server")
if tagged("old-slow-server") {
    notify { "this will succeed": }
}
```

4. If you want to set a tag on a particular resource, use the `tag` metaparameter:

```
file { "/etc/ssh/sshd_config":
    source => "puppet:///modules/admin/sshd_config",
    notify => Service["ssh"],
    tag    => "security",
}
```

5. You can also use tags to determine which parts of the manifest to apply. If you use the `--tags` option on the Puppet command line, only those classes or resources tagged with specific tags will be applied. For example, if you want to update only the Exim configuration, but not run any other parts of the manifest:

```
# puppet agent --test --tags exim
```

There's more...

You can use tags to create a collection of resources, for example if some service depends on a large number of file snippets:

```
class firewall::service {
    service { “firewall”:
        ….
    }

    File <| tag == “firewall-snippet” |> ~> Service[“firewall”]
}

class myapp {
    file { “/etc/firewall.d/myapp.conf”:
        tag => “firewall-snippet”,
        ….
    }
}
```

Here, we've specified that the `firewall` service should be notified if any `file` resource tagged `firewall-snippet` is updated. All we need to do to add a firewall config snippet for any particular app or service is to tag it `firewall-snippet`, and Puppet will do the rest.

Although we could add a `notify => Service["firewall"]` to each snippet resource, if our definition of the `firewall` service were ever to change we would have to hunt down and update all the snippets accordingly. The tag lets us encapsulate the logic in one place, making future maintenance and refactoring much easier.

Using run stages

A common requirement is to apply a certain resource before all others (for example, installing a package repository), or after all others (for example, deploying an application once its dependencies are installed). Puppet's run stages allow you to do this.

How to do it...

1. Add the following to your manifest:

```
class install_repos {
     notify { "This will be done first": }
 }

 class deploy_app {
     notify { "This will be done last": }
 }

 stage { "first": before => Stage["main"] }
 stage { "last": require => Stage["main"] }

 class { "install_repos": stage => "first" }
 class { "deploy_app": stage => "last" }
```

36.Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1303127505'
notice: This will be done first
notice: /Stage[first]/Beginning/Notify[This will be done first]/message:
defined 'message' as 'This will be done first'
notice: This will be done last
```

```
notice: /Stage[last]/End/Notify[This will be done last]/message: defined
'message' as 'This will be done last'
notice: Finished catalog run in 0.59 seconds
```

How it works...

1. We declared the classes for the things we want done first and last:

    ```
    class install_repos {
        notify { "This will be done first": }
    }

    class deploy_app {
        notify { "This will be done last": }
    }
    ```

2. Then we created a run stage named `first`:

    ```
    stage { "first": before => Stage["main"] }
    ```

The `before` parameter specifies that everything in stage `first` must be done before anything in stage `main` (the default stage).

3. Then we created a run stage named `last`:

    ```
    stage { "last": require => Stage["main"] }
    ```

The `require` parameter specifies that stage `main` must be completed before any resource in stage `last`.

4. Finally, we included the two classes `install_repos` and `deploy_app`, specifying that they should be part of stages `first` and `last` respectively:

    ```
    class { "install_repos": stage => "first" }
    class { "deploy_app": stage => "last" }
    ```

Note that we used the `class` keyword, rather than `include`, just like when we were passing parameters to classes. You can think of `stage` as a parameter that can always be passed to any class.

Puppet will now apply the stages in the following order:

- first
- main
- last

There's more...

In fact, you can define as many run stages as you like, and set up any ordering for them. This can greatly simplify a complicated manifest which otherwise would require lots of explicit dependencies between resources. If you can divide all the resources into groups A and B, and everything in A must be done before B, it's a prime candidate for using run stages.

Gary Larizza has written a helpful introduction to using run stages, with some real-world examples, at `http://glarizza.posterous.com/using-run-stages-with-puppet`

Using environments

Are your manifests environmentally friendly? If you want to test Puppet manifests before putting them into production, you can use Puppet's **environment** feature to do this. This lets you apply a different manifest depending on the environment setting of the client machine. For example, you might define the following environments:

- development
- staging
- production

You can set up environments in your `puppet.conf` file. In this example, we'll add a `development` environment, pointing to a different set of manifests.

How to do it...

1. Add the following lines to `puppet.conf`:

```
[development]
manifest = /etc/puppet/env/development/manifests/site.pp
modulepath = /etc/puppet/env/development/modules:/etc/puppet/
modules
```

How it works...

You can put your environment manifests anywhere you like, so long as you set the `manifest` parameter to point to the top-level `site.pp` file. In this example we've put the manifests for this environment in `/etc/puppet/env/development`. Similarly, you need to set `modulepath` to the location of your modules directory for that environment.

In the example above, the `modulepath` also includes `/etc/puppet/modules`; this is so that if Puppet doesn't find a module in your `development` environment, it will also look for it in the default environment. This means you only need to put the modules you're working on into the `development` environment.

The default environment is `production`, so if you run Puppet without specifying an environment, that's what you'll get.

There's more...

If you are using a version control system such as Git, your environments can be Git branches. Once you have finished testing and staging a new module, you can merge it into the Git master branch for use in production. You can read more about this strategy for using environments in R.I. Pienaar's article: `http://www.devco.net/archives/2009/10/10/puppet_environments.php`

You can specify the environment of a client machine in several ways. You can use the `--environment` switch when running Puppet:

```
# puppet agent --test --environment=development
```

Alternatively, you can specify it in the client's `puppet.conf`:

```
[main]
environment=development
```

If you are using an external node classifier script (described elsewhere in this book), this can also specify the client's environment.

You can also have a different `fileserver.conf` for each environment (see the section on configuring Puppet's file server). To do this, set the variable `fileserverconfig` for each environment in the Puppetmaster's `puppet.conf` file:

```
[development]
fileserverconfig = /etc/puppet/fileserver.conf.development

[production]
fileserverconfig = /etc/puppet/fileserver.conf.production
```

For more information, see the Puppet Labs page on using environments: `http://projects.puppetlabs.com/projects/1/wiki/Using_Multiple_Environments`

See also

- Using version control
- Using modules
- Using an external node classifier

3
Puppet Language and Style

"Elegance is not a dispensable luxury but a factor that decides between success and failure."

— Edsger Dijkstra

In this chapter we will cover:

- Using modules
- Using standard naming conventions
- Using community Puppet style
- Using embedded Ruby
- Writing manifests in pure Ruby
- Iterating over multiple items
- Writing powerful conditional statements
- Using regular expressions in `if` statements
- Using selectors and `case` statements
- Checking if values are contained in strings
- Using regular expression substitutions

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**. A module is simply a way of grouping related things: for example, a `webserver` module might include everything necessary to be a webserver: Apache configuration files, virtual host templates, and the Puppet code necessary to deploy these.

Separating things into modules makes it easier to re-use 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...

1. Find your `modulepath`; this is set in `puppet.conf` but the default value is `/etc/puppet/modules`. If you are using version control for your Puppet manifests, as I recommend you do, then use the directory in your working copy which will be deployed to `/etc/puppet/modules` instead.

   ```
   # puppet --genconfig |grep modulepath
   modulepath = /etc/puppet/modules:/usr/share/puppet/modules
   # cd /etc/puppet/modules
   ```

2. Create a directory `memcached`:

   ```
   # mkdir memcached
   ```

3. Inside this, create `manifests` and `files` directories:

   ```
   # cd memcached
   # mkdir manifests files
   ```

4. In the `manifests` directory, create the file `init.pp` with the following contents:

   ```
   import "*"
   ```

5. In the same directory, create another file `memcached.pp` with the following contents:

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

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

       service { "memcached":
   ```

```
        ensure  => running,
        enable  => true,
        require => [ Package["memcached"],
                     File["/etc/memcached.conf"] ],
    }
}
```

6. Changing to the `files` directory, create the file `memcached.conf` with the following contents:

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

7. To use your new module, add this to your node definition:

   ```
   node cookbook {
       include memcached
   }
   ```

8. Run Puppet to test the new configuration:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1300361964'
   notice: /Stage[main]/Memcached/Package[memcached]/ensure: ensure
   changed 'purged' to 'present'
   ...
   info: /Stage[main]/Memcached/File[/etc/memcached.conf]:
   Filebucketed /etc/memcached.conf to puppet with sum a977521922a151
   c959ac953712840803
   notice: /Stage[main]/Memcached/File[/etc/memcached.conf]/content:
   content changed '{md5}a977521922a151c959ac953712840803' to '{md5}f
   5c0bb01a24a5b3b86926c7b067ea6ba'
   notice: Finished catalog run in 20.68 seconds


   # service memcached status
    * memcached is running
   ```

How it works...

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:

```
MODULEPATH/
    MODULE_NAME/
        files/
        templates/
        manifests/
            init.pp
            ...
        README
```

Puppet will find and load the `init.pp` file automatically. This should import all the classes necessary for the module, as in our example:

```
import "*"
```

The `memcached` class is defined in the file `memcached.pp`, which will be imported by `init.pp`. Now we can include it on nodes:

```
include memcached
```

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

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

As we saw in the section on Puppet's file server and custom mount points, the `source` parameter above tells Puppet to look for the file in:

```
MODULEPATH/
    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. Here are a few tips which may help you on the way.

Templates

If you need to use a template as part of the module, place it in the `MODULE_NAME/templates` directory and refer to it like this:

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

Puppet will look for the file in:

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

Facts, functions, types, and providers

Modules can also contain custom facts, custom functions, and custom types and providers. For more information about these, see the chapter on 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 the section on using public modules available from Puppet Forge.

Module organization

For more details on how to organize your modules, see the Puppet Labs site: `http://projects.puppetlabs.com/projects/1/wiki/Puppet_Modules`

See also

- Configuring Puppet's file server
- Creating custom Facter facts
- Using public modules
- Creating your own resource types
- Creating your own providers

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 more true if other people need to read and work on your manifests.

How to do it...

1. Name modules after the software or service they manage: for example, `apache` or `haproxy`.
2. Name classes within modules 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 which disables Apache should be named `apache::disabled`.
4. If a node provides multiple services, have the node definition include one module or class named for each service. For example:

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

5. The module that manages users should be named `user`.
6. Within the `user` module, declare your virtual users within the class `user::virtual`.
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. Where you need to override a class for some specific node or service, inherit that class and prefix the name of the subclass with the node - for example, if your node `cartman` needs a special SSH configuration, and you want to override the `ssh` class, do it like this:

```
class cartman_ssh inherits ssh {
    [ override config here ]
}
```

9. When using Puppet to deploy 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`
   - Log rotate config snippet for Rails: `rails.logrotate`
   - Nginx vhost file for `mywizzoapp`: `mywizzoapp.vhost.nginx`
   - MySQL config for standalone server: `standalone.mysql`

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

How it works...

No explanation needed.

There's more...

The Puppet community maintains a set of **best practice** guidelines for your Puppet infrastructure which includes some hints on naming: `http://projects.puppetlabs.com/projects/1/wiki/Puppet_Best_Practice`

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.

How to do it...

1. Always quote your resource names - for example:

```
package { "exim4":
```

not

```
package { exim4:
```

Some characters like hyphens and spaces can confuse Puppet's parser, and to be on the safe side it's wise to put all names consistently in double quotes.

2. Always quote parameter values which are not reserved words in Puppet - for example:

```
name => "Nucky Thompson",

mode => "0700",

owner => "deploy",
```

but

```
ensure => installed,

enable => true,

ensure => running,w
```

3. Always include curly braces ({ }) around variable names when referring to them in strings. For example:

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

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

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

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

Very often you'll edit the file later and want to add an extra parameter after the last one, and forget to add the necessary comma!

5. When declaring a resource with a single parameter, make the declaration all on one line and with no trailing comma:

```
package { "puppet": ensure => installed }
16.Where there is more than one parameter, give each parameter its
own line:
package { "rake":
    ensure   => installed,
    provider => gem,
    require  => Package["rubygems"],
}
```

6. When declaring symlinks, use `ensure => link` like this:

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

7. To make the code easier to read, line up the parameter arrows in line with the longest parameter, like this:

```
file { "/var/www/${app}/shared/config/rvmrc":
    owner   => "deploy",
    group   => "deploy",
    content => template("rails/rvmrc"),
    require => File["/var/www/${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.

How it works...

No explanation necessary.

There's more...

The Puppet community maintains a style guide document on the Puppet Labs site: `http://projects.puppetlabs.com/projects/puppet/wiki/Style_Guide`

Using embedded Ruby

Templates are a powerful way of using embedded Ruby to help build config files dynamically and iterate over arrays, for example. But you can embed Ruby in your manifests directly without having to use a separate file, by calling the `inline_template` function.

How to do it...

1. Pass your Ruby code to `inline_template` within the Puppet manifest like this:

   ```
   cron { "nightly-job":
       command => "/usr/local/bin/nightly-job",
       hour => "0",
       minute => inline_template("<%= hostname.hash.abs % 60 %>"),
   }
   ```

How it works...

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

See also

- Using ERB templates
- Generating template files using array iteration

Writing manifests in pure Ruby

Puppet has sometimes been criticized for requiring you to write manifests in its own dedicated configuration language, rather than an existing general-purpose language such as Ruby. Not everyone considers this a drawback: the computer scientist Dennis Ritchie remarked, *"A language that doesn't have everything is actually easier to program in than some that do."*

Whatever your views, this criticism no longer applies: Puppet has experimental support for writing manifests in Ruby, which is quite usable in production even though it is still at a fairly early stage. You can mix and match Ruby and Puppet files within your manifests - Puppet will determine the language based on the file extension: `.rb` for Ruby files, `.pp` for Puppet files.

The domain-specific language (DSL) for writing manifests in Ruby looks very similar to the standard Puppet language. In this worked example I'll show you how to turn a typical Puppet manifest into Ruby. Here's the original manifest in Puppet's language:

```
class admin::exim {
    package { "exim4": ensure => installed }

    service { "exim4":
        ensure  => running,
        require => Package["exim4"],
    }

    file { "/etc/exim4/exim4.conf":
        content => template("admin/exim4.conf"),
        notify  => Service["exim4"],
        require => Package["exim4"],
    }
}
```

How to do it...

1. Create the file `/etc/puppet/modules/admin/manifests/exim.rb` with the following contents:

```
hostclass "admin::exim" do
    package "exim4", :ensure => :installed

    service "exim4",
        :ensure  => :running,
        :require => "Package[exim4]"

    file "/etc/exim4/exim4.conf",
        :content => template(["admin/exim4.conf"]),
        :notify  => "Service[exim4]",
        :require => "Package[exim4]"
end
```

2. Include this class on a node and run Puppet.

How it works...

1. The keyword `hostclass` declares a class, just like `class` in Puppet:

   ```
   hostclass "admin::exim" do
   ```

2. We then have a `do ... end` block which is the equivalent of curly braces in Puppet.
3. Resources are declared by calling a function named after the resource type: for example, `package` or `service`.

   ```
       package "exim4", :ensure => :installed
   ```

4. Parameters are passed to the function as a comma-separated list, with the parameter names quoted or given a leading colon to make them a Ruby symbol:

   ```
           :ensure  => :running,
   ```

Again built-in Puppet names such as `:installed` or `:running` are Ruby symbols.

5. When we need to refer to resources to indicate a relationship, as with `:require`, the resource identifier is given as a string with the resource type capitalized and the name in square brackets:

   ```
           :require => "Package[exim4]"
   ```

6. We can call a function like `template` by just using its name and round brackets, and passing its arguments as an array delimited by square brackets:

   ```
           :content => template(["admin/exim4.conf"]),
   ```

There's more...

I feel duty-bound to say that I don't actually advise you to use the Ruby DSL for your Puppet manifests. It's fun to experiment with, but unless there are really compelling reasons for using Ruby, I'd stick to the standard Puppet language for now. It's quite possible that in the future the Ruby DSL will become widely used; however, I doubt it. If you do want to use it, however, here are a couple of handy hints.

Variables

While you can use Ruby variables just as you normally would in a Ruby program, you can access your Puppet variables by using `scope.lookupvar` like this:

```
notify (["I am running on node %s" % scope.lookupvar("fqdn")])
```

gives:

```
notice: I am running on node cookbook.bitfieldconsulting.com
```

To set a variable so that it is in scope within your Puppet manifest, use `scope.setvar`:

```
require 'time'
scope.setvar("now", Time.now)
notify (["At the third stroke, the time sponsored by Bitfield
Consulting will be: %s" % scope.lookupvar("now")])
```

gives:

```
notice: At the third stroke, the time sponsored by Bitfield Consulting
will be: Wed Mar 23 05:58:16 -0600 2011
```

Documentation

You can find more about how to use the Ruby DSL, including more advanced topics such as virtual resources and collections, on the Puppet Labs site: `http://projects.puppetlabs.com/projects/1/wiki/Ruby_Dsl`

Ken Barber has supplied some syntax examples giving a direct comparison between Puppet and Ruby DSL constructs: `https://github.com/bobsh/puppet-rubydsl-examples`

Finally, James Turnbull has written a blog post showing a more advanced use of Ruby to connect to a MySQL server: `http://www.puppetlabs.com/blog/using-ruby-in-the-puppet-ruby-dsl/`

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 able to help. You can create an array just by putting its contents in square brackets:

```
$lunch = [ "eggs", "beans", "chips" ]
```

How to do it...

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 of instantiating lots of similar resources.

There's more...

If you thought arrays were exciting, wait till you hear about hashes.

Hashes

A **hash** is like an array, but each of the elements can be stored and looked up by name. For example:

```
$interface = { name => 'eth0',
               address => '192.168.0.1' }
notice("Interface ${interface[name]} has address
${interface[address]}")

Interface eth0 has address 192.168.0.1
```

Hash values can be anything that you can assign to a variable: strings, function calls, expressions, even other hashes or arrays.

Creating arrays with `split`

You can declare literal arrays using square brackets, like this:

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

$lunch = [ "egg", "beans", "chips" ]
lunchprint { $items: }

Lunch included egg
Lunch included beans
Lunch included chips
```

But Puppet can also create arrays for you from strings, using the `split` function, like this:

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

Lunch included egg
Lunch included beans
Lunch included chips
```

Note that `split` takes two arguments: the first is the string to split. The second 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 a string:

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

It 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 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...

1. Add the following code to your manifest:

```
if $lsbdistid == "Ubuntu" {
    notice("Running on Ubuntu")
} else {
    notice("Non-Ubuntu system detected. Please upgrade to Ubuntu
immediately.")
}
```

How it works...

Puppet treats whatever follows the `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...

You can write very complicated `if` statements in Puppet, but I recommend you don't. Very often, it's better to change your design (for example, using a template) rather than use `if`. While looking through some of my production setups for examples, I was surprised to find that I haven't used `if` at all in many thousands of lines of code. Still, your mileage may vary, so here are some more tips on using `if`.

elsif

You can add further tests using the `elsif` keyword, like this:

```
if $lsbdistid == "Ubuntu" {
    notice("Running on Ubuntu")
elsif $lsbdistid == "Debian" {
    notice("Close enough…")
} else {
    notice("Non-Ubuntu system detected. Please upgrade to Ubuntu
immediately.")
}
```

Comparisons

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

```
if $lsbdistid == "Ubuntu" {
    ...
}
```

Or you can check if they are not equal using `!=`:

```
if $lsbdistid != "CentOS" {
   ...
}
```

You can also compare numeric values using < and >:

```
if $uptime_days > 365 {
    notice("Woohoo!")
}
```

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

```
if $lsbmajdistrelease <= 9 {
    ...
}
```

Combining expressions

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

```
if ($uptime_days > 365) and ($lsbdistid == "Ubuntu") {
    ...
}

if ($role == "webserver") and ( ($datacenter == "A") or ($datacenter
== "B") ) {
    ...
}
```

See also

- Using regular expressions in `if` statements
- Checking if values are contained in strings
- Using selectors and `case` statements

Using regular expressions in 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 of comparing strings using pattern matching.

How to do it…

1. Add the following to your manifest:

```
if $lsbdistdescription =~ /LTS/ {
    notice("Looks like you are using a Long Term Support version
of Ubuntu.")
} else {
    notice("You might want to upgrade to a Long Term Support
version of Ubuntu...")
}
```

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 as a whole succeeds, the `if` expression will be true and so the code between the first set of curly braces will be executed.

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

```
if $lsbdistdescription !~ /LTS/ {
```

There's more...

Arch-hacker Jamie Zawinski once remarked:

> *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.*

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 (.*)/ {
    notice("You said '$0'. Looks like you're comparing $1 to $2!")
}
```

```
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, that 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 in the example above.

Regular expression syntax

Puppet uses a subset of Ruby regular expression syntax, so this link may be helpful if you're not already familiar with regular expressions: `http://gnosis.cx/publish/programming/regular_expressions.html`

See also

- Using regular expression substitutions

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...

1. Add the following to your manifest:

```
$systemtype = $operatingsystem ? {
    "Ubuntu" => "debianlike",
    "Debian" => "debianlike",
    "RedHat" => "redhatlike",
    "Fedora" => "redhatlike",
    "CentOS" => "redhatlike",
}

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

2. Add the following 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": {
        include redhatlike
    }
}
```

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 `$operatingsystem` to each of the possible values we have supplied: `"Ubuntu"`, `"Debian"`, and so on. These values could be regular expressions (for a partial string match, or to use wildcards, for example), but in our case we have just used literal strings. As soon as it finds a match, the selector expression returns whatever value is associated with the matching string. If the value of `$operatingsystem` is `"Fedora"`, for example, the selector expression will return the string `"redhatlike"` and so this will be assigned to the variable `$systemtype`.

case statement

Unlike selectors, the `case` statement does not return a value. `case` statements are handy where 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 class `debianlike`, or the class `redhatlike`, 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 `include debianlike` will be executed.

There's more...

Once you've got to grips with 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 matched groups and refer to them using `$1`, `$2`, etc:

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

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

Defaults

Both selectors and `case` statements let you specify a `default` value, which is chosen if none of the other options match:

```
$lunch = "Sausage and chips"
$lunchtype =  $lunch ? {
    /chips/ => "unhealthy",
    /salad/ => "healthy",
    default => "unknown",
}

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

```
Your lunch was unhealthy
```

Checking if values are contained in strings

Puppet 2.6 introduced a new kind of expression, using the `in` keyword, like this:

```
"foo" in $bar
```

This is true if the string `foo` is a substring of `$bar`. If `$bar` is an array, the expression is true if `foo` is an element of the array. If `$bar` is a hash, the expression is true if `foo` is one of the keys of `$bar`.

How to do it...

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:

```
# puppet agent --test
Debian-type operating system detected
```

How it works...

No explanation necessary.

There's more...

`in` expressions can be used not just for `if` statements or other conditionals, but anywhere an expression can be used - so, for example, you can assign the result to a variable:

```
$debianlike = $operatingsystem in [ "Debian", "Ubuntu" ]

if $debianlike {
    $ntpservice = "ntp"
} else {
    $ntpservice = "ntpd"
}
```

Using regular expression substitutions

Puppet's `regsubst` function provides an easy way to manipulate text, search and replace within strings, or extract patterns from strings. We commonly 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 IP address (the network part, assuming it's a Class C address).

How to do it...

1. Add the following to your manifest:

```
$class_c = regsubst($ipaddress, "(.*)\..*", "\1.0")
notify { $ipaddress: }
notify { $class_c: }
```

2. Run Puppet:

```
notice: 10.0.2.15
notice: 10.0.2.0
```

How it works...

`regsubst` takes at least three parameters: `source`, `pattern`, and `replacement`. In our example, we specified the `source` string as `$ipaddress`, which happens to be:

```
10.0.2.15
```

the `pattern` as:

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

and the `replacement` as:

```
\1.0
```

The `pattern` will match the whole IP address, capturing the first three octets in round brackets. The captured text will be available as `\1` for use in the `replacement` string.

The whole of the matched text (in this case the whole string) is replaced with `replacement`. This is `\1` (the captured text from the `source` string) followed by the string `.0`, which evaluates to:

```
10.0.2.0
```

There's more

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

See also

- Importing dynamic information with generate
- Getting information about the environment
- Using regular expressions in `if` statements

4
Writing Better Manifests

"An expert is someone who is one page ahead of you in the manual."

— David Knight

In this chapter we will cover:

- Using arrays of resources
- Using `define`s
- Using dependencies
- Using node inheritance
- Using class inheritance and overriding
- Passing parameters to classes
- Writing reusable, cross-platform manifests
- Getting information from the environment
- Importing dynamic information with `generate`
- Importing data from CSV files
- Passing arguments to shell commands

Introduction

Your Puppet manifest is the living documentation for your entire infrastructure. Keeping it tidy and well-organized is a great way to make it easier to maintain and understand. Puppet gives you a number of tools to do this, including:

- Arrays
- `define`s
- Dependencies
- Inheritance
- Tags
- Class parameters
- Run stages

We'll see how to use all of these and more. As you read through the chapter, try out the examples, and look through your own manifests to see where these features might help you simplify and improve your Puppet code.

Using arrays of resources

"If we wish to count lines of code, we should not regard them as lines produced but as lines spent."

— Edsger Dijkstra

Anything you can do to a resource, you can do to an array of resources. Use this idea to refactor your manifests to make them shorter and clearer.

How to do it...

1 Identify a class in your manifest where you have several instances of the same kind of resource - for example, packages:

```
package { "sudo" : ensure => installed }
package { "unzip" : ensure => installed }
package { "locate" : ensure => installed }
package { "lsof" : ensure => installed }
package { "cron" : ensure => installed }
package { "rubygems" : ensure => installed }
```

2. Group them together and replace them with a single `package` resource using an array:

   ```
   package { [ "cron",
               "locate",
               "lsof",
               "rubygems"
               "screen",
               "sudo"
               "unzip" ]:
      ensure => installed,
   }
   ```

How it works...

Most of Puppet's resource types can accept an array instead of a single name, and will create one instance for each of the elements in the array. All the parameters you provide for the resource (for example, `ensure => installed`) will be assigned to each of the new resource instances.

See also

- Iterating over multiple items

Using defines

In the previous example, we saw how to reduce redundant code by grouping identical resources into arrays. However, this technique is limited to resources where all the parameters are the same. When you have a set of resources which have some parameters in common and some different, you need to use a `define` to group them together.

How to do it...

1. Add the following to your manifest:

   ```
   define tmpfile() {
      file { "/tmp/$name":
          content => "Hello, world",
      }
   }

   tmpfile { ["a", "b", "c"]: }
   ```

2. Run Puppet:

```
 notice: /Stage[main]//Node[cookbook]/Tmpfile[a]/File[/tmp/a]/
ensure: defined content as '{md5}bc6e6f16b8a077ef5fbc8d59d0b931b9'
 notice: /Stage[main]//Node[cookbook]/Tmpfile[b]/File[/tmp/b]/
ensure: defined content as '{md5}bc6e6f16b8a077ef5fbc8d59d0b931b9'
 notice: /Stage[main]//Node[cookbook]/Tmpfile[c]/File[/tmp/c]/
ensure: defined content as '{md5}bc6e6f16b8a077ef5fbc8d59d0b931b9'
```

How it works...

You can think of a `define` as being like a cookie-cutter. It describes a pattern which Puppet can use to create lots of similar resources. Any time you declare a `tmpfile` instance in your manifest, Puppet will insert all the resources contained in the `tmpfile` definition.

In our example, the definition of `tmpfile` contains a single `file` resource, whose `content` is `"Hello, world"`, and whose `path` is `/tmp/${name}`. If you declared an instance of `tmpfile` with the name `foo` like this:

```
tmpfile { "foo": }
```

then Puppet would create a file with the path `/tmp/foo`. In other words, `${name}` in the definition will be replaced by the name of any actual instance that Puppet is asked to create. It's almost as though we created a new kind of resource: a `tmpfile`, which has one parameter: its name.

Just like with regular resources, we don't have to pass just one name: we can provide an array of names and Puppet will create a number of `tmpfile`s, as in the example.

There's more...

In the example, we created a `define` where the only parameter that varies between instances is the name. But we can add whatever parameters we want, so long as we declare them in the definition:

```
define tmpfile( $greeting ) {
   file { "/tmp/$name":
       content => $greeting,
   }
}
```

and pass values to them when we declare an instance of the resource:

```
tmpfile{ "foo": greeting => "Hello, world" }
```

You can declare multiple parameters as a comma-separated list:

```
define webapp( $domain, $path, $platform ) {
   ...
}

webapp { "mywizzoapp":
   domain   => "mywizzoapp.com",
   path     => "/var/www/apps/mywizzoapp",
   platform => "Rails",
}
```

This is a powerful technique for abstracting out everything that's common to certain resources, and keeping it in one place so that you **Don't Repeat Yourself**. In the example above, there might be many individual resources contained within `webapp`: packages, config files, source code checkouts, virtual hosts, and so on. But all of them are the same for every instance of `webapp` except the parameters we provide. These might be referenced in a template, for example, to set the domain for a virtual host.

Using dependencies

To make sure things happen in the right order, you can specify in Puppet that one resource depends on another - for example, you need to install package X before you can start the service it provides, so you would mark the service as dependent on the package. Puppet will sort out the required order to meet all the dependencies.

In some configuration management systems, resources are simply applied in the order you write them - in other words, the ordering is implicit. That's not the case with Puppet, where resources are applied in a more or less random order unless you state an explicit ordering using dependencies. Some people prefer the implicit approach, because you can write the resource definitions in the order that they need to be done, and that's the way they'll be executed.

On the other hand, in many cases the ordering of resources doesn't matter. With an implicit-style system, you can't tell whether resource A is listed before resource B because of a dependency, or because it just happens to have been listed first. That makes refactoring more difficult, as moving resources around may break some implicit dependency.

Puppet makes you do a little more work by specifying the dependencies up front, but the resulting code is clearer and easier to maintain. Let's look at an example.

How to do it...

1. Create a new file `/etc/puppet/modules/admin/manifests/ntp.pp` with the following contents:

```
class admin::ntp {
   package { "ntp":
       ensure => installed,
   }

   service { "ntp":
       ensure  => running,
       require => Package["ntp"],
   }

   file { "/etc/ntp.conf":
       source  => "puppet:///modules/admin/ntp.conf",
       notify  => Service["ntp"],
       require => Package["ntp"],
   }
}
```

2. Copy your existing `ntp.conf` file into Puppet:

```
# cp /etc/ntp.conf /etc/puppet/modules/admin/files
```

3. Add the `admin::ntp` class to your server in `nodes.pp`:

```
node cookbook {
   include admin::ntp
}
```

4. Now remove the existing `ntp.conf` file:

```
# rm /etc/ntp.conf
```

5. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1302960655'
notice: /Stage[main]/Admin::Ntp/File[/etc/ntp.conf]/ensure:
defined content as '{md5}3386aaad98dd5e0b28428966dac9e1f5'
info: /Stage[main]/Admin::Ntp/File[/etc/ntp.conf]: Scheduling
refresh of Service[ntp]
notice: /Stage[main]/Admin::Ntp/Service[ntp]: Triggered 'refresh'
from 1 events
```

```
        notice: Finished catalog run in 2.36 seconds
```

How it works...

This example demonstrates two kinds of dependency: `require`, and `notify`. In the first case, the `ntp` service requires the `ntp` package to be applied first:

```
service { "ntp":
   ensure  => running,
   require => Package["ntp"],
}
```

In the second case, the NTP config file is set to `notify` the `ntp` service; in other words, if the file changes, Puppet should restart the `ntp` service to pick up its new configuration.

```
file { "/etc/ntp.conf":
   source  => "puppet:///modules/admin/ntp.conf",
   notify  => Service["ntp"],
   require => Package["ntp"],
}
```

This implies that the service depends on the file as well as on the package, and so Puppet will be able to apply all three resources in the correct order:

```
Package["ntp"] -> File["/etc/ntp.conf"] ~> Service["ntp"]
```

In fact, this is another way to specify the same dependency chain. Adding the line above to your manifest will have the same effect as they `require` and `notify` parameters in our example (the `->` means `require`, while `~>` means `notify`. However, I prefer to use `require` and `notify` because the dependencies are defined as part of the resource, so it's easier to see what's going on. For complex chains of dependencies, though, you may want to use the `->` notation instead.

There's more...

You can also specify that a resource depends on a certain class:

```
  require => Class["my-apt-repo"]
```

You can specify dependencies not just between resources and classes, but between **collections**:

```
Yumrepo <| |> -> Package <| provider == yum |>
```

is a powerful way to express that all `yumrepo` resources should be applied before all `package` resources whose `provider` is `yum`.

Using node inheritance

Let's say you have dedicated servers hosted with three different providers: WreckSpace, GoDodgy, and VerySlow. They have different data centers and geographical locations, so you will need to make small modifications to your config for servers hosted with each provider. You have several different types of servers, but they are distributed randomly across the three providers.

One way to implement this in Puppet would be to set a variable in the node definition which tells the node where it is:

```
node webserver127 {
   $provider = "VerySlow"
   include admin::basics
   include admin::ssh
   include admin::ntp
   include puppet::client
   include backup::client
   include webserver
}

node loadbalancer5 {
   $provider = "WreckSpace"
   include admin::basics
   include admin::ssh
   include admin::ntp
   include puppet::client
   include backup::client
   include loadbalancer
}
```

As you can see, this results in a lot of duplication. It would be much easier if we simply defined a kind of node that is a "WreckSpace server", for example, and then we could create nodes which **inherit** from that node, including only the classes that determine what it does: `loadbalancer` or `webserver`.

How to do it...

1. Create a base class for all your nodes, which contains only the classes that every node has:

```
node server {
   include admin::basics
   include admin::ssh
   include admin::ntp
```

```
       include puppet::client
       include backup::client
    }
```

2. Create three different subclasses of this `server` node, each with the appropriate `provider` variable:

```
    node wreckspace_server inherits server {
       $provider = "WreckSpace"
    }

    node gododgy_server inherits server {
       $provider = "GoDodgy"
    }

    node veryslow_server inherits server {
       $provider = "VerySlow"
```

3. Now let's say you need to create a new web server in VerySlow. To do this, just inherit from `veryslow_server`:

```
    node webserver904 inherits veryslow_server {
       include webserver
    }
```

How it works...

When one node inherits from another, it picks up all the configuration that the parent node had. You can then add anything which makes this particular node different.

You can have a node inherit from a node that inherits from another node, and so on. You can't inherit from more than one node, though - so you can't have, for example:

```
 node movable_server inherits gododgy_server, veryslow_server,
wreckspace_server {
    # This won't work
 }
```

There's more...

Just as with a normal node definition, you can specify a list of node names which will all inherit the same definition:

```
node webserver1, webserver2, webserver3 inherits wreckspace_server {
   ...
}
```

Or a regular expression which will match multiple servers:

```
node /webserver\d+.veryslow.com/ inherits veryslow_server {
   ...
}
```

See also

- Using class inheritance and overriding

Using class inheritance and overriding

Just as nodes can inherit from other nodes, to save you duplicating lots of stuff for nodes which are very similar, the same idea works for classes.

For example, imagine you have a class `apache` which manages the Apache web server, and you want to set up a new Apache machine but with a slightly different config file - perhaps listening on a different port.

You could duplicate the whole of the `apache` class, except for the config file. Alternatively, you could take the config file out of the `apache` class and create two new classes, each of which includes the base `apache` class and adds a different version of the config file.

A cleaner way is to inherit from the `apache` class, but override just the config file.

Getting ready...

1. Create the directory structure for a new `apache` module:

   ```
   # mkdir /etc/puppet/modules/apache
   # mkdir /etc/puppet/modules/apache/manifests
   ```

2. Create the file `/etc/puppet/modules/apache/manifests/init.pp` with the following contents:

   ```
   import "*"
   ```

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

   ```
   class apache {
      package { "apache2-mpm-worker": ensure => installed }

      service { "apache2":
          enable  => true,
          ensure  => running,
          require => Package["apache2-mpm-worker"],
   ```

```
        }

        file { "/etc/apache2/ports.conf":
            source => "puppet:///modules/apache/port80.conf.apache",
            notify => Service["apache2"],
        }
    }
```

4. Install the Apache package, if it's not already present, and copy the included `ports.conf` file into Puppet:

   ```
   # apt-get install apache2-mpm-worker
   # cp /etc/apache2/ports.conf /etc/puppet/modules/apache/files/port80.conf.apache
   ```

5. Add the `apache` class to a node:

   ```
   node cookbook {
      include apache
   }
   ```

6. Run Puppet to verify that the manifest works.

How to do it...

1. Create a new version of `port80.conf.apache` named `port8000.conf.apache` with the following changes:

   ```
   NameVirtualHost *:8000
   Listen 8000
   ```

2. Now add a new file `/etc/puppet/modules/apache/manifests/port8000.pp` with the following contents:

   ```
   class apache::port8000 inherits apache {
      File["/etc/apache2/ports.conf"] {
          source => "puppet:///modules/apache/port8000.conf.apache",
      }
   ```

3. Change your node to include the `apache::port8000` class instead of `apache`:

   ```
   node cookbook {
      include apache::port8000
   }
   ```

4. Run Puppet to check that it makes the required changes:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   ```

```
 info: Applying configuration version '1302970905'
 --- /etc/apache2/ports.conf 2010-11-18 14:16:23.000000000 -0700
 +++ /tmp/puppet-file20110416-6165-pzeivi-0  2011-04-16
10:21:47.204294334 -0600
 @@ -5,8 +5,8 @@
 # Debian etch). See /usr/share/doc/apache2.2-common/NEWS.Debian.
gz and
 # README.Debian.gz


 -NameVirtualHost *:80
 -Listen 80
 +NameVirtualHost *:8000
 +Listen 8000


 <IfModule mod_ssl.c>
     # If you add NameVirtualHost *:443 here, you will also have
to change
 info: FileBucket adding /etc/apache2/ports.conf as {md5}38b31d203
26f3640a8dfbe1ff5d1c4ad
 info: /Stage[main]/Apache/File[/etc/apache2/ports.conf]:
Filebucketed /etc/apache2/ports.conf to puppet with sum 38b31d2032
6f3640a8dfbe1ff5d1c4ad
 notice: /Stage[main]/Apache/File[/etc/apache2/ports.conf]/
content: content changed '{md5}38b31d20326f3640a8dfbe1ff5d1c4ad'
to '{md5}41d9d446f779c55f13c5fe5a7477d943'
 info: /Stage[main]/Apache/File[/etc/apache2/ports.conf]:
Scheduling refresh of Service[apache2]
 notice: /Stage[main]/Apache/Service[apache2]: Triggered 'refresh'
from 1 events
 notice: Finished catalog run in 4.85 seconds
```

How it works...

Let's take another look at the new class:

```
class apache::port8000 inherits apache {
   File["/etc/apache2/ports.conf"] {
       source => "puppet:///modules/apache/port8000.conf.apache",
   }
}
```

You can see that after the class name we have `inherits apache`. This will make the class an exact copy of `apache`, except for the changes that follow.

The line:

```
File["/etc/apache2/ports.conf"] {
```

specifies that we want to make changes to the `file` resource named `/etc/apache2/ports.conf` in the parent class (note that `File` is capitalized, meaning that we're referring to an existing resource rather than defining a new one).

The line:

```
   source => "puppet:///modules/apache/port8000.conf.apache",
```

means that we are going to override the `source` parameter of the parent class's resource with a new value. The result will be exactly the same as if we had copied the whole class definition from `apache` but changed the value of `source`:

```
class apache {
   package { "apache2-mpm-worker": ensure => installed }

   service { "apache2":
       enable  => true,
       ensure  => running,
       require => Package["apache2-mpm-worker"],
   }

   file { "/etc/apache2/ports.conf":
       **source => "puppet:///modules/apache/port8000.conf.apache",**
       notify => Service["apache2"],
   }
}
```

There's more...

Overriding inherited classes may seem complicated at first. Once you get the idea, though, it's actually quite simple. It's a great way to make your manifests more readable because it removes lots of duplication, and focuses only on the parts which differ. Here are some more ways to use overriding.

Undefining parameters

Sometimes you don't want to change the value of a parameter, you just want to remove it altogether. To do this, use the value `undef`:

```
class apache::norestart inherits apache {
        File["/etc/apache2/ports.conf"] {
            notify => undef,
        }
    }
```

Adding extra values using +>

Similarly, instead of replacing a value, you may want to add more values to those defined in the parent class. The **plusignment** operator `+>` will do this:

```
    class apache::ssl inherits apache {
        file { "/etc/ssl/certs/cookbook.pem":
            source => "puppet:///modules/apache/cookbook.pem",
        }

        Service["apache2"] {
            require +> File["/etc/ssl/certs/cookbook.pem"],
        }
    }
```

The `+>` operator adds a value (or an array of values surrounded by square brackets) to the value defined in the parent class. In this case, what we end up with is the equivalent of this:

```
    service { "apache2":
        enable  => true,
        ensure  => running,
        require => [ Package["apache2-mpm-worker"], File["/etc/ssl/certs/
cookbook.pem"] ],
    }
```

Disabling resources

One of the most common uses for inheritance and overrides is to disable services or other resources:

```
    class apache::disabled inherits apache {
        Service["apache2"] {
            enable  => false,
            ensure  => stopped,
        }
    }
```

See also

- *Using node inheritance*
- *Passing parameters to classes*

- *Using standard naming conventions*

Passing parameters to classes

"It should be noted that no ethically-trained software engineer would ever consent to write a `DestroyBaghdad` *procedure. Basic professional ethics would instead require him to write a* `DestroyCity` *procedure, to which* `Baghdad` *could be given as a parameter."*

— Nathaniel S Borenstein

Sometimes it's very useful to parameterize some aspect of a class. For example, you might need to manage different versions of a `gem` package, and rather than making separate classes for each which differ only in the version number, or using inheritance and overrides, you can pass in the version number as a parameter.

How to do it...

1. Declare the parameter as part of the class definition:

```
class eventmachine( $version ) {
   package { "eventmachine":
       provider => gem,
       ensure   => $version,
   }
}
```

2. Then use the following syntax to include the class on a node:

```
class { "eventmachine": version => "0.12.8" }
```

How it works...

The class definition:

```
class eventmachine( $version ) {
```

is just like a normal class definition except it specifies that the class takes one parameter: `$version`. Inside the class, we've defined a `package` resource:

```
package { "eventmachine":
    provider => gem,
    ensure   => $version,
}
```

This is a `gem` package, and we're requesting to install version `$version`.

When you include the class on a node, instead of the usual syntax:

```
include eventmachine
```

there's a `class` statement:

```
class { "eventmachine": version => "0.12.8" }
```

which has the same effect, but also sets a value for the parameter `$version`.

There's more...

You can specify multiple parameters for a class:

```
class mysql( $package, $socket, $port ) {
```

and supply them in the same way:

```
class { "mysql":
   package => "percona-sql-server-5.0",
   socket  => "/var/run/mysqld/mysqld.sock",
   port    => "3306",
}
```

You can also give default values for some of your parameters:

```
class mysql( $package, $socket, $port = "3306" ) {
```

or all:

```
class mysql(
   package = "percona-sql-server-5.0",
   socket  = "/var/run/mysqld/mysqld.sock",
   port    = "3306" ) {
```

Unlike a `define`, only one instance of a parameterized class can exist on a node. So where you need to have several different instances of the resource, use a `define` instead.

See also

- Using node inheritance
- Using class inheritance and overriding

Writing reusable, cross-platform manifests

"The pessimist complains about the wind; the optimist expects it to change; the realist adjusts the sails."

— William Arthur Ward

Every system administrator dreams of a unified, homogeneous infrastructure, of identical machines all running the same version of the same OS. As in other areas of life, however, the reality is often messy and doesn't conform to the plan.

You are probably responsible for a bunch of assorted servers of varying age and architecture, running different kernels from different OS distributions, often scattered across different data centers and ISPs.

This situation should strike terror into the hearts of sysadmins of the "SSH in a `for` loop" persuasion, because executing the same commands on every server can have different, unpredictable, and even dangerous results.

We should certainly strive to bring older servers up to date and get everything as far as possible working on a single reference platform to make administration simpler, cheaper, and more reliable. But until we get there, Puppet makes coping with heterogeneous environments slightly easier.

How to do it...

1. If you have servers in different data centers which need slightly different network configuration, for example, use the node inheritance technique to encapsulate the differences:

   ```
   node wreckspace_server inherits server {
          include admin::wreckspace_specific
   }
   ```

2. Where you need to apply the same manifest to servers with different OS distributions, the main differences will probably be the names of packages and services, and the location of config files. Try to capture all these differences into a single class, using selectors to set global variables:

   ```
   $ssh_service = $operatingsystem? {
      /Ubuntu|Debian/ => "ssh",
      default         => "sshd",
   }
   ```

Then you needn't worry about the differences in any other part of the manifest; when you refer to something, use the variable in confidence that it will point to the right thing in each environment:

```
service { $ssh_service:
    ensure => running,
}
```

3. Often we need to cope with mixed architectures; this can affect the paths to shared libraries, and also may require different versions of packages. Again, try to encapsulate all the required settings in a single `architecture` class which sets global variables:

   ```
   $libdir = $architecture ? {
      x86_64  => "/usr/lib64",
      default => "/usr/lib",
   }
   ```

Then you can use these wherever an architecture-dependent value is required, in your manifests or even in templates:

```
; php.ini
[PHP]
; Directory in which the loadable extensions (modules) reside.
extension_dir = <%= libdir %>/php/modules
```

How it works...

The advantage of this approach (which could be called "top-down") is that you only need to make your choices once. The alternative, bottom-up approach, would be to have a selector or `case` statement everywhere a setting is used:

```
service { $operatingsystem? {
    /Ubuntu|Debian/ => "ssh",
    default         => "sshd" }:
    ensure => running,
}
```

This not only results in lots of duplication, but makes the code harder to read. And when a new operating system is added to the mix, you'll need to make changes throughout the whole manifest, instead of just in one place.

There's more...

If you are writing a module for public distribution (for example on Puppet Forge), you can make it much more valuable by making it as cross-platform as possible. As far as you can, test it on lots of different distributions, platforms, and architectures, and add the appropriate variables so it works everywhere.

If you use a public module and adapt it to your own environment, consider updating the public version with your changes if you think they might be helpful to other people.

Even if you are not thinking of publishing a module, bear in mind that it may be in production use for a long time and may have to adapt to many changes in the environment. If it's designed to cope with this from the start, it'll make life easier for you - or whoever ends up maintaining your code.

> *"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."*
>
> *— Dave Carhart*

See also

- Using node inheritance
- Using class inheritance and overriding
- Using public modules

Getting information about the environment

Often in a Puppet manifest, you need to know some local information about the machine you're on. **Facter** is the tool that accompanies Puppet to provide a standard way of getting information ('facts') from the environment about things like:

- operating system
- memory size
- architecture
- processor count

To see a complete list of the facts available on your system, run:

```
# facter
```

While it can be handy to get this information from the command line, the real power of Facter lies in being able to access these facts in your Puppet manifests.

How to do it...

1. Reference a Facter fact in your manifest like any other variable:

```
 notify { "This is $operatingsystem version
$operatingsystemrelease, on $architecture architecture, kernel
version $kernelversion": }
```

When Puppet runs, it will fill in the appropriate values for the current node:

```
 notice: This is Ubuntu version 10.04, on i386 architecture, kernel
version 2.6.32
```

How it works...

Facter provides an abstraction layer for Puppet, and a standard way for manifests to get information about their environment. When you refer to a fact in a manifest, Puppet will query Facter to get the current value, and insert it into the manifest.

There's more...

You can also use facts in **ERB** templates. For example, you might want to insert the node's hostname into a file, or change a config setting for an application based on the memory size of the node. When you use fact names in templates, remember that they don't need a dollar sign, because this is Ruby, not Puppet:

```
 $KLogPath <%= case kernelversion when "2.6.31" then "/var/run/
rsyslog/kmsg" else "/proc/kmsg" end %>
```

See also

- Creating custom Facter facts

Importing dynamic information with generate

Even though system administrators like to wall themselves off from the rest of the office using piles of old printers, they sometimes need to exchange information with other departments. For example, you may need to insert data into your Puppet manifests which is derived from some outside source. The `generate` function is very useful for this.

Getting ready...

1. Create the script `/usr/local/bin/latest-puppet.rb` on the Puppetmaster with the following contents:

```
#!/usr/bin/ruby

require 'open-uri'

page = open("http://www.puppetlabs.com/misc/download-options/").
read
print page.match(/stable version is ([\d\.]*)/)[1]
```

How to do it...

1. .Add the following to your manifest:

```
 $latestversion = generate("/usr/local/bin/latest-puppet.rb")
 notify { "The latest stable Puppet version is ${latestversion}.
You're using ${puppetversion}.": }
```

2. Run Puppet:

```
# puppet agent --test
notice: The latest stable Puppet version is 2.6.7. You're using
2.6.4.
```

How it works...

The `generate` function runs the specified script or program on the Puppetmaster (not the client) and returns the result - in this case, the version number of the latest stable Puppet release.

I don't recommend you run this script in production, as Puppet Labs have a habit of rearranging their web site, but you get the idea. Anything a script can do, print, fetch, or calculate - for example the results of a database query - can be brought into your manifest using `generate`.

It's worth remembering that, just as with embedded Ruby calls in templates, the `generate` is run on the Puppetmaster and not on the node which is running Puppet. I once made this mistake by calling `/bin/hostname` in a template and finding to my surprise that all my nodes were apparently named `puppet`.

When you need to get information specifically about the node, this is best done with a custom fact.

There's more...

If you need to pass arguments to the executable called by `generate`, add them as extra arguments to the function call:

```
 $latestpuppet = generate("/usr/local/bin/latest-version.rb",
"puppet")
 $latestmc = generate("/usr/local/bin/latest-version.rb",
"mcollective")
```

Puppet will try to protect you from malicious shell calls by restricting the characters you can use in a call to `generate`, so shell pipelines aren't allowed, for example. The simplest and safest thing to do is to put all your logic into a script and then call that script.

See also

- Creating custom Facter facts
- Importing data from CSV files

Importing data from CSV files

When you need to look up some value in a table, you could do it with lengthy `case` statements or selectors, but a neater way is to use the `extlookup` function. This queries an external CSV file on the Puppetmaster and returns the matching piece of data.

Grouping all such data into a single file and moving it outside the Puppet manifests makes it easier to maintain, as well as easier to share with other people: a development team can manage the things Puppet needs to know about their application, for example, by deploying a suitable CSV file as part of the release. Puppet just needs to know where to find the file, and `extlookup` will do the rest.

Getting ready...

1. Create the file `/var/www/apps/common.csv` with the following contents:

   ```
   path,/var/www/apps/%{name}
   railsversion,3
   domain,www.%{name}.com
   ```

2. Create the file `/var/www/apps/myapp.csv` with the following contents:

   ```
   railsversion,2
   ```

How to do it...

1. Add the following to your manifest:

```
$extlookup_datadir = "/var/www/apps/"
$extlookup_precedence = [ "%{name}", "common" ]

class app( $name ) {
   $railsversion = extlookup("railsversion")
   $path = extlookup("path")
   $domain = extlookup("domain")
   notify { "App data: Path ${path}, Rails version
${railsversion}, domain ${domain}": }
}

class { "app": name => "myapp" }
```

2. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1303129760'
notice: App data: Path /var/www/apps/myapp, Rails version 2,
domain www.myapp.com
notice: /Stage[main]/App/Notify[App data: Path /var/www/apps/
myapp, Rails version 2, domain www.myapp.com]/message: defined
'message' as 'App data: Path /var/www/apps/myapp, Rails version 2,
domain www.myapp.com'
notice: Finished catalog run in 0.58 seconds
```

How it works...

1. The first thing we do is define the variable `$extlookup_datadir`, which tells `extlookup` what directory to look for data files in. You would normally set this in `site.pp` or wherever you define global variables.

```
$extlookup_datadir = "/var/www/apps/"
```

2. Then we tell `extlookup` what data files to look at, in order of precedence:

```
$extlookup_precedence = [ "%{name}", "common" ]
```

This can be an array of any length, and when we make an `extlookup` query, Puppet will try each of the files in order until it finds one that has the requested value. The file names can contain variables. In this example, we've used `%{name}`, so we're expecting a variable called `$name` to be set when we call `extlookup` and Puppet will use its value as the first filename to look for.

1. Next, inside the `app` class, we call `extlookup` to get a value:

    ```
    $railsversion = extlookup("railsversion")
    ```

The `extlookup` machinery now looks for a CSV file to read the data from. It looks in the `$extlookup_datadir` directory (in this case `/var/www/apps`) for a file named `%{name}.csv` (in this case `myapp.csv`). So it reads the file `/var/www/apps/myapp.csv` which contains:

```
railsversion,2
```

We've found the required value (`2`), so `extlookup` returns it.

1. The next `extlookup` call isn't so lucky:

    ```
    $path = extlookup("path")
    ```

Again, `extlookup` looks first in `myapp.csv`, but it doesn't find a value matching `path`. So it moves on to the next file listed in `$extlookup_precedence`, which is `common.csv`:

```
path,/var/www/apps/%{name}
railsversion,3
domain,www.%{name}.com
```

Thankfully, this does match, so Puppet returns the value `/var/www/apps/%{name}`, which in this case evaluates to `/var/www/apps/myapp`.

You can see that this allows us to have a set of default values in `common.csv` which each app may choose to override in its own `myapp.csv` file. `extlookup` will keep on querying the files listed in `$extlookup_precedence` until it finds the value requested. As `myapp.csv` is listed first, any setting in it will take precedence over settings in `common.csv`.

There's more...

You can also specify default values in the `extlookup` call, to be used if no suitable data is found in the CSV files:

```
    $path = extlookup("path", "/var/www/misc")
```

You can also specify a CSV file to be consulted first, before anything in `$extlookup_precedence`:

```
    $path = extlookup("path", "/var/www/misc", "paths")
```

This will look in `paths.csv` for the data, and if it doesn't find it, will move on to the files listed in `$extlookup_precedence` as usual.

The values in your CSV files can also refer to variables, as we did here:

```
domain,www.%{name}.com
```

You can use any variable that's in scope, including Facter facts:

```
domain,%{fqdn}
```

R.I. Pienaar's article "Complex data and Puppet" is an excellent introduction to `extlookup`: `http://www.devco.net/archives/2009/08/31/complex_data_and_puppet.php`

Jordan Sissel has written about configuring your whole infrastructure using `extlookup`: `http://sysadvent.blogspot.com/2010/12/day-12-scaling-operability-with-truth.html`

See also

- *Importing dynamic information*
- *Creating custom Facter facts*

Passing arguments to shell commands

If you need to insert values into a command line, they often need to be quoted, especially if they contain spaces. The `shellquote` function will take any number of arguments, including arrays, and quote each of the arguments and return them all as a space-separated string which you can pass to commands.

In this example, we would like to set up an `exec` resource which will rename a file, but both the source and the target name contain spaces, so they need to be correctly quoted in the command line.

How to do it...

1. Add the following to your manifest:

```
$source = "Hello Jerry"
$target = "Hello... Newman"
$argstring = shellquote( $source, $target )
$command = "/bin/mv ${argstring}"
notify { $command: }
```

2. Run Puppet:

```
notice: /bin/mv "Hello Jerry" "Hello... Newman"
```

How it works...

1. First we define the `$source` and `$target` variables, which are the two filenames we want to use in the command line.

   ```
   $source = "Hello Jerry"
   $target = "Hello... Newman"
   ```

2. Then we call `shellquote` to concatenate these variables into a quoted, space-separated string.

   ```
   $argstring = shellquote( $source, $target )
   ```

3. Then we put together the final command line:

   ```
   $command = "/bin/mv ${argstring}"
   ```

4. The result is:

   ```
   /bin/mv "Hello Jerry" "Hello... Newman"
   ```

5. This command line can now be run with an `exec` resource.

What would happen if we didn't use `shellquote`?

```
$source = "Hello Jerry"
$target = "Hello... Newman"
$command = "/bin/mv ${source} ${target}"
notify { $command: }

    notice: /bin/mv Hello Jerry Hello... Newman
```

This won't work because `mv` expects space-separated arguments, so will interpret this as a request to move three files `Hello`, `Jerry`, and `Hello...` into a directory named `Newman`, which probably isn't what we want.

5
Working with Files and Packages

"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."

— `Gerald Weinberg`

In this chapter we will cover:

- Making quick edits to `config` files
- Using Augeas to automatically edit config files
- Using dependencies
- Building config files using snippets
- Using ERB templates
- Using array iteration in templates
- Installing packages from a third-party repository
- Setting up an APT package repository
- Setting up a gem repository
- Building packages automatically from source
- Comparing package versions

Making quick edits to config files

Did you know Puppet can do keyhole surgery? Often we don't want to have to put a whole config file into Puppet just to add one setting - especially if the file is managed by someone else and we can't overwrite it. What would be useful is a simple recipe to add a line to a config file if it's not already present.

You can use an `exec` resource to do jobs like this: this example, from the Puppet Labs wiki, shows how to use `exec` to append a line to a text file.

How to do it...

1. Create the file `/etc/puppet/manifests/utils.pp` with the following contents:

   ```
   define append_if_no_such_line($file, $line) {
       exec { "/bin/echo '$line' >> '$file'":
           unless => "/bin/grep -Fx '$line' '$file'"
       }
   }
   ```

2. Add this line to `/etc/puppet/manifests/site.pp`:

   ```
   import "utils.pp"
   ```

3. Now add this to your manifest:

   ```
   append_if_no_such_line { "enable-ip-conntrack":
       file => "/etc/modules",
       line => "ip_conntrack",
   }
   ```

4. Run Puppet:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1303649606'
   notice: /Stage[main]//Node[cookbook]/Append_if_no_such_
   line[enable-ip-conntrack]/Exec[/bin/echo 'ip_conntrack' >> '/etc/
   modules']/returns: executed successfully
   notice: Finished catalog run in 1.22 seconds
   ```

How it works...

The exec resource will append the specified text in $line to the file $file:

```
    exec { "/bin/echo '$line' >> '$file'":
```

unless it's already present:

```
        unless => "/bin/grep -Fx '$line' '$file'"
```

This append_if_no_such_line resource is now available for you to use in your manifest. In this example, we've used it to ensure that the /etc/modules file (which specifies kernel modules to load at boot time) contains the line:

```
ip_conntrack
```

There's more...

You can use similar defines to perform other minor operations on text files. For example, this will enable you to search and replace within a file:

```
define replace_matching_line( $match, $replace ) {
  exec { "/usr/bin/ruby -i -p -e 'sub(%r{$match}, \"$replace\")'
$name":
    onlyif => "/bin/grep -E '$match' $name",
    logoutput => on_failure,
  }
}

replace_matching_line { "/etc/apache2/apache2.conf":
    match   => "LogLevel .*",
    replace => "LogLevel debug",
}
```

See also

- Using Augeas to automatically edit config files

Using Augeas to automatically edit config files

Of course, the great thing about standards is that there are so many of them. Sometimes it seems like every application has its own subtly different config file format, and writing regular expressions to parse and modify all of them can be a tiresome business.

Thankfully, **Augeas** is here to help. Augeas is a tool which aims to simplify working with different config file formats, by presenting them all as a simple tree of values. Puppet's Augeas support allows you to create `augeas` resources which can make the required config changes intelligently and automatically.

Getting ready...

1. Create the file `/etc/puppet/modules/admin/manifests/augeas.pp` with the following contents:

```
class admin::augeas {
    package { [ "augeas-lenses",
                "augeas-tools",
                "libaugeas0",
                "libaugeas-ruby1.8" ]:
        ensure => "present"
    }
}
```

2. Include this class on a node:

```
node cookbook {
    include admin::augeas
}
```

3. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1303657095'
notice: /Stage[main]/Admin::Augeas/Package[augeas-tools]/ensure:
ensure changed 'purged' to 'present'
notice: Finished catalog run in 21.96 seconds
```

How to do it...

1. Create the file `/etc/puppet/modules/admin/manifests/ipforward.pp` with the following contents:

```
class admin::ipforward {
    augeas { "enable-ip-forwarding":
        context => "/files/etc/sysctl.conf",
        changes => [
            "set net.ipv4.ip_forward 1",
        ],
        }
    }
```

2. Include this class on a node:

```
node cookbook {
    include admin::augeas
    include admin::ipforward
}
```

3. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1303729376'
notice: /Stage[main]/Admin::Ipforward/Augeas[enable-ip-
forwarding]/returns: executed successfully
notice: Finished catalog run in 3.53 seconds
```

4. Check that the setting has been correctly applied:

```
# sysctl -p |grep forward
net.ipv4.ip_forward = 1
```

How it works…

1. We declare an `augeas` resource named `enable-ip-forwarding`:

```
    augeas { "enable-ip-forwarding":
```

2. We specify that we want to make changes in the context of the file `/etc/sysctl.conf`:

```
        context => "/files/etc/sysctl.conf",
```

3. The parameter `changes` is passed an array of settings that we want to make (in this case only one):

```
        changes => [
            "set net.ipv4.ip_forward 1",
        ],
```

In general Augeas changes take the form:

```
set <parameter> <value>
```

Augeas uses a set of translation files called **lenses** to enable it to write these settings in the appropriate format for the given config file. In this case, the setting will be translated into a line like this in `/etc/sysctl.conf`:

```
net.ipv4.ip_forward=1
```

There's more...

I've chosen `/etc/sysctl.conf` as the example because it can contain a wide variety of kernel settings and you may want to change these settings for all sorts of different purposes and in different Puppet classes. You might want to enable IP forwarding, as in the example, for a router class, but you might also want to tune the value of `net.core.somaxconn` for a load-balancer class.

This means that simply Puppetising the `/etc/sysctl.conf` file and distributing it as a text file won't work, because you might have several different and conflicting versions, depending on the setting you want to modify. Augeas is the right solution here because you can define `augeas` resources in different places which modify the same file, and they won't conflict.

Augeas is a powerful tool which ships with lenses for most of the standard Linux config files, and you can write your own for rare or proprietary config formats if you need to manage these. For more about using Puppet and Augeas, see the page on the Puppet Labs wiki: `http://projects.puppetlabs.com/projects/1/wiki/Puppet_Augeas`

Building config files using snippets

How do you eat an elephant? One bite at a time. Sometimes you have a situation where you want to build up a single config file from various snippets managed by different classes. For example, you might have two or three services which require `rsync` modules to be configured, so you can't distribute a single `rsyncd.conf`. Although you could use Augeas, there's a simple way to concatenate config snippets together into a single file using an `exec`.

How to do it...

1. Create the file `/etc/puppet/modules/admin/manifests/rsyncdconf.pp` with the following contents:

```
class admin::rsyncdconf {
    file { "/etc/rsyncd.d":
        ensure => directory,
    }

    exec { "update-rsyncd.conf":
        command     => "/bin/cat /etc/rsyncd.d/*.conf > /etc/
rsyncd.conf",
        refreshonly => true,
    }
}
```

2. Add the following to your manifest:

```
class myapp::rsync {
    include admin::rsyncdconf

    file { "/etc/rsyncd.d/myapp.conf":
        ensure  => present,
        source  => "puppet:///modules/myapp/myapp.rsync",
        require => File["/etc/rsyncd.d"],
        notify  => Exec["update-rsyncd.conf"],
    }
}

include myapp::rsync
```

3. Create the file `/etc/puppet/modules/myapp/files/myapp.rsync` with the following contents:

```
[myapp]
    uid = myappuser
    gid = myappuser
    path = /opt/myapp/shared/data
    comment = Data for myapp
    list = no
    read only = no
    auth users = myappuser
```

4. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1303731804'
notice: /Stage[main]/Admin::Rsyncdconf/File[/etc/rsyncd.d]/ensure:
created
notice: /Stage[main]/Myapp::Rsync/File[/etc/rsyncd.d/myapp.conf]/
ensure: defined content as '{md5}e1e57cf38bb88a7b4f2fd6eb1ea2823a'
info: /Stage[main]/Myapp::Rsync/File[/etc/rsyncd.d/myapp.conf]:
Scheduling refresh of Exec[update-rsyncd.conf]
notice: /Stage[main]/Admin::Rsyncdconf/Exec[update-rsyncd.conf]:
Triggered 'refresh' from 1 events
notice: Finished catalog run in 1.01 seconds
```

How it works...

The `admin::rsyncdconf` class creates a directory for `rsync` config snippets to be placed into:

```
file { "/etc/rsyncd.d":
    ensure => directory,
}
```

When you create a config snippet (such as in `myapp::rsync`), all you need to do is `require` the directory:

```
require => File["/etc/rsyncd.d"],
```

and `notify` the `exec` that updates the main config file:

```
notify  => Exec["update-rsyncd.conf"],
```

This `exec` will then be run every time one of these snippets is updated:

```
exec { "update-rsyncd.conf":
    command     => "/bin/cat /etc/rsyncd.d/*.conf > /etc/rsyncd.conf",
    refreshonly => true,
}
```

which will concatenate all the snippets in `/etc/rsyncd.d` into `rsyncd.conf`.

There's more...

This is a useful pattern whenever you have a service like `rsync` that has a single config file which may contain distinct snippets. In effect, it gives you the functionality of Apache's `conf.d` or PHP's `php-ini.d` directories.

See also

- Using tags

Using ERB templates

A template is a text file with a college degree. Anywhere you might deploy a text file using Puppet, you can use a template instead. In the simplest case, a template can just be a static text file. More usefully, you can insert variables into it using **ERB** (embedded Ruby) syntax. For example:

```
<%= name %>, this is a very large drink.
```

If the template is used in a context where the variable `$name` contains `Zaphod Beeblebrox`, the template will evaluate to:

```
Zaphod Beeblebrox, this is a very large drink.
```

This simple technique is very useful for generating lots of files which only differ in the values of one or two variables: for example, virtual host files. In this example, we'll use an ERB template to generate an Apache virtual host definition.

Getting ready...

1. Create an Apache module, if you don't have one:

   ```
   # mkdir /etc/puppet/modules/apache
   # mkdir /etc/puppet/modules/apache/templates
   # mkdir /etc/puppet/modules/apache/manifests
   ```

2. Create the file `/etc/puppet/modules/apache/manifests/init.pp` with the following contents:

   ```
   import "*"
   ```

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

   ```
   class apache {
       package { "apache2-mpm-worker": ensure => installed }

       service { "apache2":
           enable  => true,
           ensure  => running,
           require => Package["apache2-mpm-worker"],
       }
   }
   ```

How to do it...

1. Create the file `/etc/puppet/modules/apache/manifests/site.pp` with the following contents:

   ```
   class apache::site( $sitedomain ) {
       include apache

       file { "/etc/apache2/sites-available/${sitedomain}.conf":
           content => template("apache/vhost.erb"),
           require => Package["apache2-mpm-worker"],
       }
   ```

```
    file { "/etc/apache2/sites-enabled/${sitedomain}.conf":
        ensure  => symlink,
        target  => "/etc/apache2/sites-available/${sitedomain}.
conf",
        require => Package["apache2-mpm-worker"],
        notify  => Service["apache2"],
  }
}
```

2. Create the file `/etc/puppet/modules/apache/templates/vhost.erb` with the following contents:

```
<VirtualHost *:80>
    ServerName <%= sitedomain %>
    ServerAdmin admin@<%= sitedomain %>
    DocumentRoot /var/www/<%= sitedomain %>
    ErrorLog logs/<%= sitedomain %>-error_log
    CustomLog logs/<%= sitedomain %>-access_log common

    <Directory /var/www/<%= sitedomain %>>
        Allow from all
        Options +Includes +Indexes +FollowSymLinks
        AllowOverride all
    </Directory>
</VirtualHost>

<VirtualHost *:80>
    ServerName www.<%= sitedomain %>
    Redirect 301 / http://<%= sitedomain %>/
</VirtualHost>
```

3. Add this to a node:

```
class { "apache::site": sitedomain => "keithlard.com" }
```

4. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1304761207'
notice: /Stage[main]/Apache::Site/File[/etc/apache2/sites-
available/keithlard.com.conf]/ensure: defined content as '{md5}f2a
558c02beeaed4beb7da250821b663'
notice: /Stage[main]/Apache::Site/File[/etc/apache2/sites-enabled/
keithlard.com.conf]/ensure: created
```

```
info: /Stage[main]/Apache::Site/File[/etc/apache2/sites-enabled/
keithlard.com.conf]: Scheduling refresh of Service[apache2]
notice: /Stage[main]/Apache/Service[apache2]: Triggered 'refresh'
from 1 events
notice: Finished catalog run in 8.06 seconds
```

How it works...

The `apache::site` class takes a parameter `sitedomain` which is the domain name for the virtual host we want to create: in this case, `keithlard.com`.

It then uses the `vhost.erb` template to generate an appropriate Apache virtual host definition. Wherever the variable `$sitedomain` is referenced in the template, for example:

```
ServerName <%= sitedomain %>
```

Puppet will replace it with the corresponding value:

```
ServerName keithlard.com
```

The beauty of the template system is that you could create multiple sites, all using the same virtual host template:

```
class { "apache::site": sitedomain => "bitfieldconsulting.com" }
class { "apache::site": sitedomain => "cribbagecorner.com" }
class { "apache::site": sitedomain => "cornishceremonies.com" }
class { "apache::site": sitedomain => "mosquito-mojito.com" }
```

and if you want to make a slight change to the configuration for all sites (for example, changing the admin email address) you can do it once in the template and Puppet will update all virtual hosts accordingly.

There's more...

In the example, we only used one variable in the template, but you can have as many as you like. These can also be facts:

```
ServerName <%= fqdn %>
```

or Ruby expressions:

```
MAILTO=<%= emails.join(',') %>
```

or any Ruby code you want:

```
ServerAdmin <%= sitedomain == 'coldcomfort.com' ? 'seth@coldcomfort.
com' : 'flora@poste.com' %>
```

See also

- Using array iteration in templates

Using array iteration in templates

In the previous example we saw that you can use Ruby to interpolate different values in templates depending on the result of an expression. You can also use a loop to generate content based on, for example, the elements of an array:

How to do it...

1. Add the following to your manifest:

```
$ipaddresses = [ '192.168.0.1',
                 '158.43.128.1',
                 '10.0.75.207' ]

file { "/tmp/addresslist.txt":
    content => template("admin/addresslist.erb")
}
```

2. Create the file `/etc/puppet/modules/admin/templates/addresslist.erb` with the following contents:

```
<% ipaddresses.each do |ip| -%>
IP address <%= ip %> is present.
<% end -%>
```

3. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1304766335'
notice: /Stage[main]//Node[cookbook]/File[/tmp/addresslist.txt]/
ensure: defined content as '{md5}7ad1264ebdae101bb5ea0afef474b3ed'
notice: Finished catalog run in 0.64 seconds
```

4. Check the contents of the generated file:

```
# cat /tmp/addresslist.txt
IP address 192.168.0.1 is present.
IP address 158.43.128.1 is present.
IP address 10.0.75.207 is present.
```

How it works...

1. In the first line of the template, we reference the array `ipaddresses`, and call its `each` method:

   ```
   <% ipaddresses.each do |ip| -%>
   ```

2. In Ruby, this creates a loop which will execute once for each element of the array. Each time round the loop, the variable `ip` will be set to the value of the current element.
3. In our example, the `ipaddresses` array contains three elements, so the following line will be executed three times, once for each element:

   ```
   IP address <%= ip %> is present.
   ```

4. This will result in three output lines:

   ```
   IP address 192.168.0.1 is present.
   IP address 158.43.128.1 is present.
   IP address 10.0.75.207 is present.
   ```

5. The final line ends the loop:

   ```
   <% end -%>
   ```

6. Note that the first and last lines end with `-%>`, instead of just `%>` as we saw before. The effect of the - is to suppress the newline that would otherwise be generated, giving us an unwanted blank line in the file.

There's more...

Templates can also iterate over hashes, or arrays of hashes:

```
$interfaces = [ { name => 'eth0',
                  ip   => '192.168.0.1' },
                { name => 'eth1',
                  ip   => '158.43.128.1' },
                { name => 'eth2',
                  ip   => '10.0.75.207' } ]

<% interfaces.each do |interface| -%>
Interface <%= interface['name'] %> has the address <%= interface['ip']
%>.
<% end -%>

Interface eth0 has the address 192.168.0.1.
Interface eth1 has the address 158.43.128.1.
Interface eth2 has the address 10.0.75.207.
```

See also

- Using ERB templates

Installing packages from a third-party repository

Most often you will want to install packages from the main distribution repo, so a simple `package` resource will do:

```
package { "exim4": ensure => installed }
```

Sometimes, though, you need a package which is only found in a third-party repository (an Ubuntu PPA, for example). Or it might be that you need a more recent version of a package than that provided by the distribution, which is available from a third party.

On a manually administered machine, you would normally do this by adding the repo source configuration to `/etc/apt/sources.list.d` (and, if necessary, a GPG key for the repo) before installing the package. We can automate this process easily with Puppet.

How to do it...

1. Add the following to your manifest:

```
package { "python-software-properties": ensure => installed }

exec { "/usr/bin/add-apt-repository ppa:mathiaz/puppet-backports":
    creates => "/etc/apt/sources.list.d/mathiaz-puppet-backports-
lucid.list",
    require => Package["python-software-properties"],
}
```

2. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1304773240'
notice: /Stage[main]//Node[cookbook]/Exec[/usr/bin/add-apt-
repository ppa:mathiaz/puppet-backports]/returns: executed
successfully
notice: Finished catalog run in 5.97 seconds
```

How it works...

1. The `python-software-properties` package provides the command `add-apt-repository`, which simplifies the process of adding extra repos:

   ```
   package { "python-software-properties": ensure => installed }
   ```

2. We then call this command in the `exec` to add the required configuration:

   ```
   exec { "/usr/bin/add-apt-repository ppa:mathiaz/puppet-backports":
   ```

3. So that the `exec` is not run every time Puppet runs, we specify a file that the command creates, so that Puppet will skip the `exec` if this file already exists:

   ```
       creates => "/etc/apt/sources.list.d/mathiaz-puppet-backports-
   lucid.list",
   ```

You might want to combine this with purging unwanted repo definitions in `/etc/apt/sources.list.d`, as described in the section on recursive file resources.

See also

- Distributing directory trees using recursive file resources

Setting up an APT package repository

"We will control the horizontal. We will control the vertical."

- The Outer Limits

Running your own package repository has several advantages. You can distribute your own packages with it. You can control the versions of upstream or third-party packages that you put into it. And you can locate it close to where your servers are, to avoid the problem of slow or unreliable mirror sites.

Even if you don't need to create your own packages, you may want to download the required versions of your critical dependency packages and store them in your own repo, thus preventing any surprises when things change upstream (for example, your distro version could reach end-of-life and the repos turned off).

It also makes it easier to auto-update packages within Puppet. You may occasionally need to update a package (for example, when a security update is available), so it's convenient to specify `ensure => latest` in the package definition. But when you don't control the repo, this puts you at risk of an unexpected upgrade which breaks something in your system.

Your own repo gives you the best of both worlds: you can auto-update the package in Puppet, but since it comes from your repo, a new version will only be available when you put one there. You can test the version from upstream before making it available in your production repo.

Getting ready...

You will need the `apache` module from the section on 'Using ERB templates', so create this if you don't already have it.

In the example, I've called the repo `packages.bitfieldconsulting.com`, because that's what mine is called. You'll probably want to use a different name, so replace it throughout the example with the name of your repo.

How to do it...

1. Create a new `repo` module:

   ```
   # mkdir /etc/puppet/modules/repo
   # mkdir /etc/puppet/modules/repo/manifests
   # mkdir /etc/puppet/modules/repo/files
   ```

2. Create the file `/etc/puppet/modules/repo/manifests/init.pp` with the following contents:

   ```
   import "*"
   ```

3. Create the file `/etc/puppet/modules/repo/manifests/bitfield-server.pp` with the following contents:

   ```
   class repo::bitfield-server {
       include apache

       package { "reprepro": ensure => installed }

       file { [ "/var/apt",
                "/var/apt/conf" ]:
           ensure => directory,
       }

       file { "/var/apt/conf/distributions":
           source  => "puppet:///modules/repo/distributions",
           require => File["/var/apt/conf"],
       }

       file { "/etc/apache2/sites-available/apt-repo":
           source  => "puppet:///modules/repo/apt-repo.conf",
   ```

```
        require => Package["apache2-mpm-worker"],
    }

    file { "/etc/apache2/sites-enabled/apt-repo":
        ensure  => symlink,
        target  => "/etc/apache2/sites-available/apt-repo",
        require => File["/etc/apache2/sites-available/apt-repo"],
        notify  => Service["apache2"],
    }
}
```

4. Create the file `/etc/puppet/modules/repo/files/distributions` with the following contents:

```
Origin: Bitfield Consulting
Label: bitfield
Suite: stable
Codename: lucid
Architectures: amd64 i386
Components: main non-free contrib
Description: Custom and cached packages for Bitfield Consulting
45.Create the file /etc/puppet/modules/repo/files/apt-repo.conf
with the following contents:
<VirtualHost *:80>
    DocumentRoot /var/apt
    ServerName packages.bitfieldconsulting.com
    ErrorLog /var/log/apache2/packages.bitfieldconsulting.com.
error.log

    LogLevel warn

    CustomLog /var/log/apache2/packages.bitfieldconsulting.com.
access.log combined
    ServerSignature On

    # Allow directory listings so that people can browse the
repository from their browser too
    <Directory "/var/apt">
        Options Indexes FollowSymLinks MultiViews
        DirectoryIndex index.html
        AllowOverride Options
        Order allow,deny
        allow from all
    </Directory>

    # Hide the conf/ directory for all repositories
    <Directory "/var/apt/conf">
        Order allow,deny
        Deny from all
        Satisfy all
    </Directory>

    # Hide the db/ directory for all repositories
    <Directory "/var/apt/db">
        Order allow,deny
        Deny from all
        Satisfy all
    </Directory>
</VirtualHost>
```

5. Add the following to the manifest for a node:

```
include repo::bitfield-server
```

6. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1304775601'
notice: /Stage[main]/Repo::Bitfield-server/File[/var/apt]/ensure:
created
notice: /Stage[main]/Repo::Bitfield-server/File[/var/apt/conf]/
ensure: created
notice: /Stage[main]/Repo::Bitfield-server/File[/var/apt/conf/
distributions]/ensure: defined content as '{md5}65dc791b876f53318a
35fcc42c770283'
notice: /Stage[main]/Repo::Bitfield-server/Package[reprepro]/
ensure: created
notice: /Stage[main]/Repo::Bitfield-server/File[/etc/apache2/
sites-enabled/apt-repo]/ensure: created
notice: /Stage[main]/Repo::Bitfield-server/File[/etc/apache2/
sites-available/apt-repo]/ensure: defined content as '{md5}2da4686
957e5acf49220047fe6f6e6e1'
info: /Stage[main]/Repo::Bitfield-server/File[/etc/apache2/sites-
enabled/apt-repo]: Scheduling refresh of Service[apache2]
notice: /Stage[main]/Apache/Service[apache2]: Triggered 'refresh'
from 1 events
notice: Finished catalog run in 16.32 seconds
```

How it works...

Actually, you don't need very much to be an APT repository. It works over HTTP, so you just need an Apache virtual host. You can put the actual package files anywhere you like, as long as there is a `conf/distributions` file which will give APT information about the repo.

1. The first part of the `bitfield-server` class makes sure we have Apache set up:

   ```
   class repo::bitfield-server {
       include apache
   ```

2. The `reprepro` tool is useful for managing the repo itself (for example, adding new packages):

   ```
       package { "reprepro": ensure => installed }
   ```

3. We create the root directory of the repo in `/var/apt`, along with the `conf/distributions` file:

   ```
       file { [ "/var/apt",
                "/var/apt/conf" ]:
           ensure => directory,
       }

       file { "/var/apt/conf/distributions":
           source  => "puppet:///modules/repo/distributions",
           require => File["/var/apt/conf"],
       }
   ```

4. The remainder of the class deploys the Apache virtual host file to enable it to serve requests on `packages.bitfieldconsulting.com`:

   ```
   file { "/etc/apache2/sites-available/apt-repo":
       source  => "puppet:///modules/repo/apt-repo.conf",
       require => Package["apache2-mpm-worker"],
   }

   file { "/etc/apache2/sites-enabled/apt-repo":
       ensure  => symlink,
       target  => "/etc/apache2/sites-available/apt-repo",
       require => File["/etc/apache2/sites-available/apt-repo"],
       notify  => Service["apache2"],
   }
   ```

There's more...

Of course, a repo isn't much good without any packages in it. In this section we'll see how to add packages, and also how to configure machines to download packages from your repo.

Adding packages

To add a package to your repo, download it and then use `reprepro` to add it to the repo:

```
# cd /tmp
# wget http://archive.ubuntu.com/ubuntu/pool/main/n/ntp/ntp_4.2.4p8+dfsg-1ubuntu2.1_i386.deb
# cd /var/apt
# reprepro includedeb lucid /tmp/ntp_4.2.4p8+dfsg-1ubuntu2.1_i386.deb
Exporting indices...
```

Configuring nodes to use the repo

1. Create the file `/etc/puppet/modules/repo/manifests/bitfield.pp` with the following contents (replacing the IP address with that of your repo server):

```
class repo::bitfield {
    host { "packages.bitfieldconsulting.com":
        ip     => "10.0.2.15",
        ensure => present,
        target => "/etc/hosts",
    }

    file { "/etc/apt/sources.list.d/bitfield.list":
        content => "deb http://packages.bitfieldconsulting.com/
lucid main\n",
        require => Host["packages.bitfieldconsulting.com"],
        notify  => Exec["bitfield-update"],
    }

    exec { "bitfield-update":
        command     => "/usr/bin/apt-get update",
        require     => File["/etc/apt/sources.list.d/bitfield.
list"],
        refreshonly => true,
    }
}
```

If you have a DNS server or control of your DNS zone, you can skip the host entry.

2. Apply this class to a node:

   ```
   node cookbook {
       include repo::bitfield
   }
   ```

3. Test whether the `ntp` package shows up as available from your repo:

   ```
   # apt-cache madison ntp
          ntp | 1:4.2.4p8+dfsg-1ubuntu2.1 | http://us.archive.ubuntu.
   com/ubuntu/ lucid-updates/main Packages
          ntp | 1:4.2.4p8+dfsg-1ubuntu2.1 | http://packages.
   bitfieldconsulting.com/ lucid/main Packages
          ntp | 1:4.2.4p8+dfsg-1ubuntu2 | http://us.archive.ubuntu.
   com/ubuntu/ lucid/main Packages
          ntp | 1:4.2.4p8+dfsg-1ubuntu2 | http://us.archive.ubuntu.
   com/ubuntu/ lucid/main Sources
          ntp | 1:4.2.4p8+dfsg-1ubuntu2.1 | http://us.archive.ubuntu.
   com/ubuntu/ lucid-updates/main Sources
   ```

Signing your packages

For production use, you should sign your packages and repo with a GPG key; for information about how to set this up, see Sander Marechal's useful article on setting up and managing APT repositories: `http://www.jejik.com/articles/2006/09/setting_up_and_managing_an_apt_repository_with_reprepro/`

Setting up a gem repository

It's every system administrator's dream: yet another incompatible packaging system. If you manage Ruby or Rails applications, you'll need to deal with Rubygems. Maintaining your own gem repository has many of the same advantages as having an APT repo: you can control availability and package versions, and you can also use it to distribute your own gems if you need to.

How to do it...

1. Create the file `/etc/puppet/modules/repo/manifests/gem-server.pp` with the following contents:

   ```
   class repo::gem-server {
       include apache

       file { "/etc/apache2/sites-available/gemrepo":
           source  => "puppet:///modules/repo/gemrepo.conf",
   ```

```
            require => Package["apache2-mpm-worker"],
            notify  => Service["apache2"],
        }

        file { "/etc/apache2/sites-enabled/gemrepo":
            ensure  => symlink,
            target  => "/etc/apache2/sites-available/gemrepo",
            require => File["/etc/apache2/sites-available/gemrepo"],
            notify  => Service["apache2"],
        }

        file { "/var/gemrepo":
            ensure => directory,
        }
    }
```

2. Create the file `/etc/puppet/modules/repo/files/gemrepo.conf` with the following contents:

```
<VirtualHost *:80>
    ServerAdmin john@bitfieldconsulting.com
    ServerName gems.bitfieldconsulting.com
    ErrorLog logs/gems.bitfieldconsulting.com-error_log
    CustomLog logs/gems.bitfieldconsulting.com-access_log common

    Alias / /var/gemrepo/
    <Location />
        Options Indexes
    </Location>
</VirtualHost>
57.Add the following to your manifest:
node cookbook {
    include repo::gem-server
}
```

3. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1304949279'
notice: /Stage[main]/Repo::Gem-server/File[/etc/apache2/sites-
available/gemrepo]/ensure: defined content as '{md5}ae1fd948098f14
503de02441d02a825d'
info: /Stage[main]/Repo::Gem-server/File[/etc/apache2/sites-
available/gemrepo]: Scheduling refresh of Service[apache2]
notice: /Stage[main]/Repo::Gem-server/File[/etc/apache2/sites-
enabled/gemrepo]/ensure: created
info: /Stage[main]/Repo::Gem-server/File[/etc/apache2/sites-
enabled/gemrepo]: Scheduling refresh of Service[apache2]
notice: /Stage[main]/Apache/Service[apache2]: Triggered 'refresh'
from 2 events
notice: /Stage[main]/Repo::Gem-server/File[/var/gemrepo]/ensure:
created
notice: Finished catalog run in 6.52 seconds
```

How it works…

The principle is exactly the same as in the APT repo example. We define a directory where the gem repository will live, and a virtual host definition in Apache to enable it to serve requests for `gems.bitfieldconsulting.com`.

There's more…

Again, your gem repo will be more useful if you put something in it. We'll find out how to do that below, and also how to configure your nodes to access the gem repo.

Adding gems

Adding new gems to your repo is simple. Put the gem file in `/var/gemrepo/gems` and run this command in the `/var/gemrepo` directory:

```
# gem generate_index
```

Using the gem repo

As with the APT repo, make sure that your nodes know about the hostname `gems.bitfieldconsulting.com`, either by deploying a host entry with Puppet, or configuring it in DNS.

Then you can specify a package in Puppet as follows:

```
package { "json":
    provider => "gem",
    source => "http://gems.bitfieldconsulting.com",
}
```

Building packages automatically from source

Tarballs can seriously damage your health. While using a distro or third-party package, or rolling your own package, is always preferable to building software from source, sometimes it has to be done. Creating Debian packages (or any other flavor of packages) can be a lengthy and error-prone process, and there may not always be time or budget available to do this.

If you have to build a program from source, Puppet can at least help with this process. The general procedure is to automate what you would otherwise do manually:

- Download the source tarball
- Unpack the tarball
- Configure and build the program
- Install the program

In this example we'll build OpenSSL from source (though for production you should use the distro package, but it makes a useful demonstration).

How to do it...

1. Add the following to your manifest:

   ```
   exec { "build-openssl":
       cwd       => "/root",
       command   => "/usr/bin/wget ftp://ftp.openssl.org/source/
   openssl-0.9.8p.tar.gz && /bin/tar xvzf openssl-0.9.8p.tar.gz && cd
   openssl-0.9.8p && ./Configure linux-generic32 && make install",
       creates   => "/usr/local/ssl/bin/openssl",
       logoutput => on_failure,
       timeout   => 0,
   }
   ```

2. Run Puppet (it may take a while):

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1304954159'
   notice: /Stage[main]//Node[cookbook]/Exec[build-openssl]/returns:
   executed successfully
   notice: Finished catalog run in 554.00 seconds
   ```

How it works...

The `exec` command is in five separate stages, delimited by `&&` operators. This means that should any subcommand fail, the whole command will stop and fail. It's a useful construct where you want to make sure each subcommand has succeeded before going on to the next.

1. The first stage downloads the source tarball:

   ```
   /usr/bin/wget ftp://ftp.openssl.org/source/openssl-0.9.8p.tar.gz
   ```

2. The second stage unpacks it:

   ```
   /bin/tar xvzf openssl-0.9.8p.tar.gz
   ```

3. The third stage changes working directory to the source tree:

   ```
   cd openssl-0.9.8p
   ```

4. The fourth stage runs the configure script (this is usually where you will need to specify any options or customisations):

   ```
   ./Configure linux-generic32
   ```

5. The final stage builds and installs the software:

   ```
   make install
   ```

6. So that this lengthy process isn't run every time Puppet runs, we specify a file that the build creates:

   ```
   creates   => "/usr/local/ssl/bin/openssl",
   ```

If you need to force a rebuild for whatever reason, remove this file.

1. Things don't always compile first time; in case of problems, we specify the `logoutput` parameter which will show us what the build process is complaining about:

   ```
       logoutput => on_failure,
   ```

2. Finally, because the compilation may take a while, we set a zero `timeout` parameter (Puppet times out `exec` commands after 5 minutes by default):

   ```
       timeout   => 0,
   ```

There's more...

If you have to build quite a few packages from source, it may be worth converting the recipe above into a `define`, so that you can use more or less the same code to build each package.

Comparing package versions

Package version numbers are odd things. They look like decimal numbers, but they're not - a version number is often in the form `2.6.4`, for example. If you need to compare one version number with another, you can't do a straightforward string comparison: `2.6.4` would be interpreted as greater than `2.6.12`. And a numeric comparison won't work because they're not valid numbers.

Puppet's `versioncmp` function comes to the rescue. If you pass it two things that look like version numbers, it will compare them and return a value indicating which is the greater:

```
versioncmp( A, B )
```

returns:

- 0 if A and B are equal
- Greater than 1 if A is higher than B
- Less than 0 if A is less than B

How to do it...

1. Add the following to your manifest:

   ```
   $app_version = "1.2.2"
   $min_version = "1.2.10"

   if versioncmp( $app_version, $min_version ) >= 0 {
       notify { "Version OK": }
   } else {
       notify { "Upgrade needed": }
   }
   ```

2. Run Puppet:

   ```
   notice: Upgrade needed
   ```

3. Now change the value of `$app_version`:

   ```
   $app_version = "1.2.14"
   ```

4. Run Puppet again:

   ```
   notice: Version OK
   ```

How it works…

We've specified that the minimum acceptable version (`$min_version`) is `1.2.10`. So in the first example, we want to compare it with an `$app_version` of `1.2.2`. A simple alphabetic comparison of these two strings (in Ruby, for example) would give the wrong result, but `versioncmp` correctly determines that `1.2.2` is less than `1.2.10` and alerts us that we need to upgrade.

In the second example, `$app_version` is now `1.2.14` which `versioncmp` correctly recognises as greater than `$min_version` and so we get the message `Version OK`.

6 Users and Other Resources

"How good the design is doesn't matter near as much as whether the design is getting better or worse. If it is getting better, day by day, I can live with it forever. If it is getting worse, I will die."

— `Kent Beck`

In this chapter we will cover:

- Using virtual resources
- Managing users with virtual resources
- Managing users' SSH access
- Managing users' customization files
- Efficiently distributing cron jobs
- Running a command when a file is updated
- Using `host` resources
- Using multiple file sources
- Distributing directory trees using recursive file resources
- Cleaning up old files
- Using schedules with resources
- Auditing resources
- Temporarily disabling resources
- Managing timezones

Using virtual resources

What are virtual resources and why do we need them? Let's look at a typical situation where virtual resources might come in useful.

You are responsible for two applications, FaceSquare and Twitstagram. Both are web apps running on Apache. The definition for FaceSquare might look something like this:

```
class app::facesquare {
    package { "apache2-mpm-worker": ensure => installed }

    ...
}
```

The definition for Twitstagram might look like this:

```
class app::twitstagram {
    package { "apache2-mpm-worker": ensure => installed }

    ...
}
```

All is well until you need to consolidate both apps onto a single server:

```
node micawber {
    include app::facesquare
    include app::twitstagram
}
```

Now Puppet will complain because you tried to define two resources with the same name: `apache2-mpm-worker`.

```
err: Could not retrieve catalog from remote server: Error 400 on SERVER:
Duplicate definition: Package[apache2-mpm-worker] is already defined in
file /etc/puppet/modules/app/manifests/facesquare.pp at line 2; cannot
redefine at /etc/puppet/modules/app/manifests/twitstagram.pp:2 on node
cookbook.bitfieldconsulting.com
```

You could remove the duplicate package definition from one of the classes, but then it would fail if you tried to include the app class on another server that didn't already have Apache.

You can get round this problem by putting the Apache package in its own class and then using `include apache`; Puppet doesn't mind you including the same class multiple times. But this has the disadvantage that every potentially conflicting resource must have its own class.

Virtual resources to the rescue. A virtual resource is just like a normal resource, except that it starts with an `@` character:

```
@package { "apache2-mpm-worker": ensure => installed }
```

You can think of it as being like an 'FYI' resource: I'm just telling you about this resource, and I don't actually want you to do anything about it yet. Puppet will read and remember virtual resource definitions, but won't actually create the resource until you say so.

To create the resource, use the `realize` function

```
realize( Package["apache2-mpm-worker"] )
```

You can call `realize` as many times as you want on the resource and it won't result in a conflict. So virtual resources are the way to go when several different classes all require the same resource and they may need to co-exist on the same node.

How to do it...

1. Create a new module `app`:

   ```
   # mkdir -p /etc/puppet/modules/app/manifests
   ```

2. Create the file `/etc/puppet/modules/app/manifests/init.pp` with the following contents:

   ```
   import "*"
   ```

3. Create the file `/etc/puppet/modules/app/manifests/facesquare.pp` with the following contents:

   ```
   class app::facesquare {
       realize( Package["apache2-mpm-worker"] )
   }
   ```

4. Create the file `/etc/puppet/modules/app/manifests/twitstagram.pp` with the following contents:

   ```
   class app::twitstagram {
       realize( Package["apache2-mpm-worker"] )
   }
   ```

5. Create the file `/etc/puppet/modules/admin/manifests/virtual-packages.pp` with the following contents:

   ```
   class admin::virtual-packages {
       @package { "apache2-mpm-worker": ensure => installed }
   }
   ```

6. Include the following on the node:

   ```
   node cookbook {
       include admin::virtual-packages
       include app::facesquare
       include app::twitstagram
   }
   ```

7. Run Puppet.

How it works...

You define the package as a virtual resource in one place, the `admin::virtual-packages` class. All nodes can include this class and you can put all your virtual packages in it. None of them will actually be installed on a node until you call `realize`.

```
class admin::virtual-packages {
    @package { "apache2-mpm-worker": ensure => installed }
}
```

Every class that needs the Apache package can call `realize` on this virtual resource:

```
class app::twitstagram {
    realize( Package["apache2-mpm-worker"] )
}
```

Puppet knows, because you made the resource virtual, that you intended multiple references to the same package, and didn't just accidentally create two resources with the same name. So it does the right thing.

There's more...

To realize virtual resources you can also use the **collection** syntax:

```
Package <| title = "apache2-mpm-worker" |>
```

The advantage of this syntax is that you're not restricted to the resource name; you could also use a tag, for example:

```
Package <| tag = "security" |>
```

Or you can just specify all instances of the resource type, by leaving the query section blank:

```
Package <| |>
```

See also

- Managing users with virtual resources

Managing users with virtual resources

Users are an excellent example of where virtual resources can come in handy. Consider the following setup. You have three users: John, Graham, and Steven. To simplify administration of a large number of machines, you have defined classes for two kinds of users: developers and sysadmins. All machines need to include sysadmins, but only some machines need developer access.

```
node server {
       include user::sysadmins
}

node webserver inherits server {
       include user::developers
}
```

John is a sysadmin, and Steven a developer, but Graham is both, so needs to be in both groups. This will cause a conflict on a `webserver` as we end up with two definitions of the user `graham`.

To avoid this situation, the Puppet community has adopted the practice of making all users virtual, defined in a single class `user::virtual` which every machine includes, and then realizing the users where they are needed.

How to do it...

1. Create a `user` module:

   ```
   # mkdir -p /etc/puppet/modules/user/manifests
   ```

2. Create the file `/etc/puppet/modules/user/manifests/init.pp` with the following contents:

   ```
   import "*"
   ```

3. Create the file `/etc/puppet/modules/app/manifests/virtual.pp` with the following contents:

   ```
   class user::virtual {
       @user { "john": }
       @user { "graham": }
       @user { "steven": }
   }
   ```

4. Create the file `/etc/puppet/modules/app/manifests/developers.pp` with the following contents:

   ```
   class user::developers {
       realize( User["graham"],
                User["steven"] )
   }
   ```

5. Create the file `/etc/puppet/modules/app/manifests/sysadmins.pp` with the following contents:

   ```
   class user::sysadmins {
       realize( User["john"],
                User["graham"] )
   }
   6.Add the following to a node:
   include user::virtual
   include user::sysadmins
   include user::developers
   ```

6. Run Puppet:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1305554239'
   notice: /Stage[main]/User::Virtual/User[john]/ensure: created
   notice: /Stage[main]/User::Virtual/User[steven]/ensure: created
   notice: /Stage[main]/User::Virtual/User[graham]/ensure: created
   notice: Finished catalog run in 2.36 seconds
   ```

How it works...

Every node should include the `user::virtual` class, as part of your basic housekeeping configuration which is inherited by all servers. This class will define all users in your organization or site. This should also include any users who exist only to run applications or services (such as `apache` or `git`, for example).

You can then organize your users into groups (not in the sense of UNIX groups, but perhaps as different teams or job roles) - `developers` and `sysadmins` is a typical example. The class for a group will realize whichever users are included in it:

```
class user::sysadmins {
    realize( User["john"],
             User["graham"] )
}
```

You can then include these groups wherever they are needed, without worrying about conflicts caused by multiple definitions of the same user.

See also

- Using virtual resources
- Managing users' customization files

Managing users' SSH access

Commonly-accepted best practice for access control to servers is to use named accounts with passphrase-protected SSH keys. Puppet makes this easy to manage thanks to the built-in `ssh_authorized_key` type.

To combine this with virtual users, as described in the previous section, you can create a `define` which includes both the `user` and the `ssh_authorized_key`. This will also come in useful for adding customization files and other per-user resources.

How to do it...

1. Change the `user::virtual` class that you created in the section on managing users with virtual resources, to the following:

```
class user::virtual {
    define ssh_user( $key ) {
        user { $name:
            ensure     => present,
            managehome => true,
         }

        ssh_authorized_key { "${name}_key":
            key  => $key,
            type => "ssh-rsa",
            user => $name,
        }
    }

    @ssh_user { "phil":
        key =>
aC1yc2EAAAABIwAAAIEA3ATqENg+GWACa2BzeqTdGnJhNoBer8x6pfWkzNzeM8Zx7/
2Tf2pl7kHdbsiTXEUawqzXZQtZzt/j3Oya+PZjcRpWNRzprSmd2UxEEPTqDw9LqY5S
2B8og/NyzWaIYPsKoatcgC7VgYHplcTbzEhGu8BsoEVBGYu3IRy5RkAcZik=",
    }
}
```

2. Include the following on a node:

   ```
   realize( User::Virtual::Ssh_user["phil"] )
   ```

3. Run Puppet:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1305561740'
   notice: /Stage[main]/User::Virtual/User::Virtual::Ssh_user[phil]/
   User[phil]/ensure: created
   notice: /Stage[main]/User::Virtual/User::Virtual::Ssh_user[phil]/
   Ssh_authorized_key[phil_key]/ensure: created
   notice: Finished catalog run in 1.04 seconds
   ```

How it works...

We've created a new `define` called `ssh_user` which includes both the `user` resource itself, and the associated `ssh_authorized_key`.

```
define ssh_user( $key ) {
    user { $name:
        ensure     => present,
        managehome => true,
     }

    ssh_authorized_key { "${name}_key":
        key  => $key,
        type => "ssh-rsa",
        user => $name,
    }
}
```

Then we create a virtual instance of `ssh_user` for the user `phil`:

```
@ssh_user { "phil":
    key =>
B3NzaC1yc2EAAAABIwAAAIEA3ATqENg+GWACa2BzeqTdGnJhNoBer8x6pfWkzNzeM8Zx7/
```

```
2Tf2pl7kHdbsiTXEUawqzXZQtZzt/j3Oya+PZjcRpWNRzprSmd2UxEEPTqDw9LqY5S2B8o
g/NyzWaIYPsKoatcgC7VgYHplcTbzEhGu8BsoEVBGYu3IRy5RkAcZik=",
}
```

Recall that because the resource is virtual, Puppet will take note of it but won't actually create anything until `realize` is called.

Finally, we added this to the node:

```
realize( User::Virtual::Ssh_user["phil"] )
```

This actually creates the user and the `authorized_keys` file containing the user's public key.

There's more...

To use this idea with the organization of users into group classes that we saw in the previous section, modify the classes like this:

```
class user::sysadmins {
    search User::Virtual

    realize( Ssh_user["john"],
             Ssh_user["graham"] )
}
```

The `search User::Virtual` is just to save on clutter; it allows you to refer to `Ssh_user` directly without prefixing it with `User::Virtual::` every time.

If you get an error like:

```
err: /Stage[main]/User::Virtual/User::Virtual::Ssh_user[graham]/Ssh_
authorized_key[graham_key]: Could not evaluate: No such file or directory
- /home/graham/.ssh
```

it may be because you previously created the `graham` user without having Puppet manage the home directory; in this situation Puppet will not automatically create the `.ssh` directory for the `authorized_keys` file. Run:

```
# userdel graham
```

and run Puppet again to fix the problem.

Managing users' customization files

Users, like cats, often feel the need to mark their territory. Unlike cats, users tend to customize their shell environments, terminal colours, aliases, and so forth. This is usually achieved by a number of **dotfiles** in their home directory: for example, `.bash_profile`.

You can add this to your Puppet-based user management by modifying the `user::virtual::ssh_user` class so that it can optionally include any dotfiles which are present in the Puppet repo.

How to do it...

1. Modify the `user::virtual` class as follows:

```
class user::virtual {
    define user_dotfile( $username ) {
        file { "/home/${username}/.${name}":
            source => "puppet:///modules/user/${username}-
${name}",
            owner  => $username,
            group => $username,
        }
    }

    define ssh_user( $key, $dotfile = false ) {
        user { $name:
            ensure     => present,
            managehome => true,
         }

        ssh_authorized_key { "${name}_key":
            key  => $key,
            type => "ssh-rsa",
            user => $name,
        }

        if $dotfile {
            user_dotfile { $dotfile:
                username => $name,
            }
        }
    }

    @ssh_user { "john":
        key     =>
aC1yc2EAAAABIwAAAIEA3ATqENg+GWACa2BzeqTdGnJhNoBer8x6pfWkzNzeM8Zx7/
2Tf2pl7kHdbsiTXEUawqzXZQtZzt/j3Oya+PZjcRpWNRzprSmd2UxEEPTqDw9LqY5S
2B8og/NyzWaIYPsKoatcgC7VgYHplcTbzEhGu8BsoEVBGYu3IRy5RkAcZik=",
        dotfile => [ "bashrc", "bash_profile" ],
    }
}
```

2. Create the file `/etc/puppet/modules/user/files/john-bashrc` with the following contents:

   ```
   export PATH=$PATH:/var/lib/gems/1.8/bin
   ```

3. Create the file `/etc/puppet/modules/user/files/john-bash_profile` with the following contents:

   ```
   . ~/.bashrc
   ```

4. Run Puppet.

How it works...

We've added a new `define`, `user_dotfile`. This will be called once for each dotfile the user wants to have. In the example, `john` has two dotfiles: `.bashrc` and `.bash_profile`. These are declared as follows:

```
@ssh_user { "john":
    key      => ...
    dotfile => [ "bashrc", "bash_profile" ],
}
```

You can supply either a single dotfile, or a list of them in array form, as above.

For each dotfile, `user_dotfile` will look for a corresponding source file in the `modules/user/files` directory. For example, with the `bashrc` dotfile, Puppet will look for:

```
modules/user/files/john-bashrc
```

This will be copied to the node as:

```
/home/john/.bashrc
```

See also

- Managing users with virtual resources

Efficiently distributing cron jobs

When you have many servers executing the same cron job, it's usually a good idea not to run them all at the same time. If the jobs all access a common server, it may put too much load on that server, and even if they don't, all the servers will be busy at once, which may affect their capacity to provide other services.

Puppet's `inline_template` function allows us to use some Ruby logic to set different runtimes for the job depending on the hostname.

How to do it...

1. Add the following to a node:

```
define cron_random( $command, $hour ) {
    cron { $name:
        command => $command,
        minute  => inline_template("<%= (hostname+name).hash.abs %
60 %>"),
        hour    => $hour,
        ensure  => "present",
    }
}

cron_random { "hello-world":
    command => "/bin/echo 'Hello world'",
    hour => 2,
}

cron_random { "hello-world-2":
    command => "/bin/echo 'Hello world'",
    hour => 1,
}
```

2. .Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1305713506'
notice: /Stage[main]//Node[cookbook]/Cron_random[hello-world]/
Cron[hello-world]/ensure: created
notice: /Stage[main]//Node[cookbook]/Cron_random[hello-world-2]/
Cron[hello-world-2]/ensure: created
notice: Finished catalog run in 1.07 seconds
```

How it works...

We want to choose a 'random' minute for each cron job; that is, not genuinely random (or it would change every time Puppet runs), but more or less guaranteed to be different for each cron job on each host.

We can do this by using Ruby's `hash` method which computes a numerical value from any object, in this case a string. The value will be the same each time, so although the value looks random, it will not change when Puppet runs again.

`hash` will generate a large integer, and we want values between 0 and 59, so we use the Ruby `%` (modulus) operator to restrict the result to this range. Although there are only 60 possible values, the hash function is designed to produce as uniform an output as possible, so there should be very few collisions and the `minute` values should be well-distributed.

We want the value to be different for identical jobs on different machines, so we use the hostname in computing the hash value. However, we also want the value to be different for different jobs on the same machine, so we combine the hostname with the `name` variable, which will be the name of the cron job (`hello-world`, for example).

There's more...

In this example we only randomized the minute of the cron job, and supplied the hour as part of the definition. If you sometimes need to specify the day of the week as well, you could add it as an optional parameter for `cron_random` with a default value:

```
define cron_random( $command, $hour, $weekday = "*" ) {
```

If you wanted to also randomize the hour (for example, for jobs which could run at any time of day and need to be distributed across all 24 hours evenly) you could modify `cron_random` as follows:

```
hour    => inline_template("<%= (hostname+name).hash.abs % 24 %>"),
```

See also

- Running Puppet from cron

Running a command when a file is updated

It's a very common pattern to have Puppet take some action whenever a particular file is updated. For example, in the `rsync` config snippet example, each snippet file called an `exec` to update the main `rsyncd.conf` file when it changed.

An `exec` resource will normally be run every time Puppet runs, unless you specify one of the following parameters:

- `creates`
- `onlyif`
- `unless`
- `refreshonly => true`

The `refreshonly` parameter means that the `exec` should only be run if it receives a `notify` from another resource (such as a file, for example).

Getting ready...

1. Install the `nginx` package:

   ```
   # apt-get install nginx
   ```

How to do it...

1. Create a new module `nginx` with the usual directory structure:

   ```
   # mkdir /etc/puppet/modules/nginx
   # mkdir /etc/puppet/modules/nginx/files
   # mkdir /etc/puppet/modules/nginx/manifests
   ```

2. Create the file `/etc/puppet/modules/nginx/manifests/init.pp` with the following contents:

   ```
   import "*"
   ```

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

   ```
   class nginx {
       package { "nginx": ensure => installed }

       service { "nginx":
           enable => true,
           ensure => running,
       }

       exec { "reload nginx":
           command     => "/usr/sbin/service nginx reload",
           require     => Package["nginx"],
           refreshonly => true,
       }
   ```

```
        file { "/etc/nginx/nginx.conf":
            source  => "puppet:///modules/nginx/nginx.conf",
            notify  => Exec["reload nginx"],
        }
    }
```

4. Copy the `nginx.conf` file into the new module:

   ```
   cp /etc/nginx/nginx.conf /etc/puppet/modules/nginx/files
   ```

5. Add the following to your manifest:

   ```
   include nginx
   ```

6. Make a test change to Puppet's copy of the `nginx.conf` file:

   ```
   # echo \# >>/etc/puppet/modules/nginx/files/nginx.conf
   ```

7. Run Puppet:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1303745502'
   --- /etc/nginx/nginx.conf   2010-02-15 00:16:47.000000000 -0700
   +++ /tmp/puppet-file20110425-31239-158xcst-0    2011-04-25
   09:39:49.586322042 -0600
   @@ -48,3 +48,4 @@
    #         proxy      on;
    #     }
    # }
   +#
   info: FileBucket adding /etc/nginx/nginx.conf as {md5}7bf139588b5e
   cd5956f986c9c1442d44
   info: /Stage[main]/Nginx/File[/etc/nginx/nginx.conf]: Filebucketed
   /etc/nginx/nginx.conf to puppet with sum 7bf139588b5ecd5956f986c9c
   1442d44
   notice: /Stage[main]/Nginx/File[/etc/nginx/nginx.conf]/content:
   content changed '{md5}7bf139588b5ecd5956f986c9c1442d44' to '{md5}d
   28d08925174c3f6917a78797c4cd3cc'
   info: /Stage[main]/Nginx/File[/etc/nginx/nginx.conf]: Scheduling
   refresh of Exec[reload nginx]
   notice: /Stage[main]/Nginx/Exec[reload nginx]: Triggered 'refresh'
   from 1 events
   notice: Finished catalog run in 1.69 seconds
   ```

How it works...

With most services, you'd simply define a `service` resource which gets a `notify` from the config file. This causes Puppet to restart the service in order that it can pick up the changes.

However, Nginx sometimes doesn't restart properly, especially when restarted by Puppet, and so I cooked up this remedy for one site to have Puppet run `/etc/init.d/nginx reload` instead of restarting it. Here's how it works.

The `exec` resource has the `refreshonly` parameter set to `true`:

```
exec { "reload nginx":
    command     => "/usr/sbin/service nginx reload",
    require     => Package["nginx"],
    refreshonly => true,
}
```

so it will only run if it receives a `notify`.

The config file resource supplies the necessary `notify` if it's changed:

```
file { "/etc/nginx/nginx.conf":
    source  => "puppet:///modules/nginx/nginx.conf",
    notify  => Exec["reload nginx"],
}
```

Whenever Puppet needs to update this file, it will also run the `exec`, which will call:

```
/usr/sbin/service nginx reload
```

to pick up the changes.

There's more...

You can use a similar pattern anywhere some action needs to be taken every time a resource is updated. Possible uses might include:

- triggering service reloads
- running a syntax check before restarting a service
- concatenating config snippets
- running tests
- chaining `exec`s

If you have several commands which all need to be run when a single file is updated, it might be easier to have the commands all `subscribe` to the file, rather than have the file `notify` the commands. The effect is the same.

Using host resources

It's common practice to move machines around, especially on cloud infrastructure, so the IP of a particular machine may change quite often. Because of this, it's obviously a bad idea to hard-code IP addresses into your configuration. Where one machine needs to access another - for example, an app server accessing a database server - it's better to use a hostname than an IP address.

But how to map names to IP addresses? This is often done with DNS, but small organizations may not have a DNS server, and large organizations may make it so time-consuming and bureaucratic to implement DNS changes that no-one bothers. Also, DNS information can propagate to machines at different times, so to ensure quick and consistent address updates one approach is to use local `/etc/hosts` entries controlled by Puppet.

How to do it...

1. .Add the following to your manifest:

   ```
   host { "www.bitfieldconsulting.com":
       ip     => "109.74.195.241",
       target => "/etc/hosts",
       ensure => present,
   }
   ```

2. Run Puppet:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1305716418'
   notice: /Stage[main]//Node[cookbook]/Host[www.bitfieldconsulting.
   com]/ensure: created
   info: FileBucket adding /etc/hosts as {md5}977bf5811de978b7f041301
   9e77b4abe
   notice: Finished catalog run in 0.21 seconds
   ```

How it works...

No explanation necessary.

There's more...

Organizing your host resources into classes can be helpful. For example, you could put the host resources for all your DB servers into one class called `admin::dbhosts` which is included by all web servers.

Where machines may need to be defined in multiple classes (for example, a database server might also be a repository server), virtual resources can solve this problem. For example, you could define all your hosts as virtual in a single class:

```
class admin::allhosts {
    @host { "db1.bitfieldconsulting.com":
        ...
    }
}
```

and then realize the hosts you need in the various classes:

```
class admin::dbhosts {
    realize( Host["db1.bitfieldconsulting.com"] )
}

class admin::repohosts {
    realize( Host["db1.bitfieldconsulting.com"] )
}
```

Using multiple file sources

A neat feature of Puppet's `file` resource is that you can specify multiple `source`s for the file. Puppet will look for each of them in order. If the first isn't found, it moves on to the next, and so on. You can use this to specify a default substitute if the particular file isn't present, or even a series of increasingly generic substitutes.

How to do it...

1. Add this class to your manifest:

```
class mysql::app-config( $app ) {
    file { "/etc/my.cnf":
        source  => [ "puppet:///modules/admin/${app}.my.cnf",
                     "puppet:///modules/admin/generic.my.cnf", ],
    }
}
```

2. Create the file `/etc/puppet/modules/admin/files/minutespace.my.cnf` with the following contents:

   ```
   # MinuteSpace config file
   ```

3. Create the file `/etc/puppet/modules/admin/files/generic.my.cnf` with the following contents:

   ```
   # Generic config file
   ```

4. Add the following to a node:

   ```
   class { "mysql::app-config": app => "minutespace" }
   ```

5. Run Puppet:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1305897071'
   notice: /Stage[main]/Mysql::App-config/File[/etc/my.cnf]/ensure:
   defined content as '{md5}24f04b960f4d33c70449fbc4d9f708b6'
   notice: Finished catalog run in 0.35 seconds
   ```

6. Check that Puppet has deployed the app-specific config file:

   ```
   # cat /etc/my.cnf
   # MinuteSpace config file
   ```

7. Now change the node definition to:

   ```
       class { "mysql::app-config": app => "shreddit" }
   ```

8. Run Puppet again:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1305897864'
   --- /etc/my.cnf 2011-05-20 13:17:56.006239489 +0000
   +++ /tmp/puppet-file20110520-15575-1icobgs-0    2011-05-20
   13:24:25.030296062 +0000
   @@ -1 +1 @@
   -# MinuteSpace config file
   +# Generic config file
   info: FileBucket adding /etc/my.cnf as {md5}24f04b960f4d33c70449fb
   c4d9f708b6
   info: /Stage[main]/Mysql::App-config/File[/etc/my.cnf]:
   ```

```
Filebucketed /etc/my.cnf to puppet with sum 24f04b960f4d33c70449fb
c4d9f708b6

notice: /Stage[main]/Mysql::App-config/File[/etc/my.cnf]/content:
content changed '{md5}24f04b960f4d33c70449fbc4d9f708b6' to '{md5}b
3a6e744c3ab78dfb20e46ff55f6c33c'

notice: Finished catalog run in 0.93 seconds
```

How it works...

We've defined the `/etc/my.cnf` file as having two sources:

```
file { "/etc/my.cnf":
    source  => [ "puppet:///modules/admin/${app}.my.cnf",
                 "puppet:///modules/admin/generic.my.cnf", ],
}
```

The value of `$app` will be passed in by anyone using the class. So in the first example, we passed in a value of `minutespace`:

```
class { "mysql::app-config": app => "minutespace" }
```

So Puppet will look first of all for `modules/admin/files/minutespace.my.cnf`. This file exists, so it will be used. So far, so normal.

Then we change the value of `app` to `shreddit`. Puppet now looks for `modules/admin/files/shreddit.my.cnf`. This doesn't exist, so Puppet tries the next listed source: `modules/admin/files/generic.my.cnf`. This does exist, so it will be deployed.

There's more...

You can use this trick anywhere you have a `file` resource. For example, some nodes might need machine-specific config, but not others, so you could do something like:

```
file { "/etc/stuff.cfg":
    source => [ "puppet:///modules/stuff/${hostname}.cfg",
                "puppet:///modules/stuff/generic.cfg" ],
}
```

Then you put the normal configuration in `generic.cfg`. If machine `cartman` needs a special config, just put it in the file `cartman.cfg`. This will be used in preference to the generic file because it is listed first in the array of sources.

See also

- Passing parameters to classes

Distributing directory trees using recursive file resources

"To understand recursion, you must first understand recursion."

— Saying

When you find yourself deploying several files with Puppet, all to the same directory, it might be worth considering a recursive file resource instead. If you set the `recurse` parameter on a directory, Puppet will copy the directory to the node along with its contents and all its subdirectories:

```
file { "/usr/lib/nagios/plugins/custom":
    source => "puppet:///modules/nagios/plugins",
    require => Package["nagios-plugins"],
    recurse => true,
}
```

How to do it...

1. Create a suitable directory tree in the Puppet repo:

   ```
   # mkdir /etc/puppet/modules/admin/files/tree
   # mkdir /etc/puppet/modules/admin/files/tree/a
   # mkdir /etc/puppet/modules/admin/files/tree/b
   # mkdir /etc/puppet/modules/admin/files/tree/c
   # mkdir /etc/puppet/modules/admin/files/tree/a/1
   ```

2. Add the following to your manifest:

   ```
   file { "/tmp/tree":
       source  => "puppet:///modules/admin/tree",
       recurse => true,
   }
   ```

3. Run Puppet:

   ```
   # puppet agent --test
   info: Retrieving plugin
   ```

```
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1304768523'
notice: /Stage[main]//Node[cookbook]/File[/tmp/tree]/ensure:
created
notice: /File[/tmp/tree/a]/ensure: created
notice: /File[/tmp/tree/a/1]/ensure: created
notice: /File[/tmp/tree/b]/ensure: created
notice: /File[/tmp/tree/c]/ensure: created
notice: Finished catalog run in 1.25 seconds
```

How it works…

If a `file` resource has the `recurse` parameter set on it, and it is a directory, Puppet will deploy not only the directory itself, but all its contents (including subdirectories and their contents). This is a great way to put a whole tree of files onto a node, or to quickly create a large number of paths using a single resource.

There's more…

Sometimes you want to deploy files to an existing directory, but remove any files which aren't managed by Puppet. For example, in Ubuntu's `/etc/apt/sources.list.d` directory, you might want to make sure there are no files present which don't come from Puppet.

The `purge` parameter will do this for you. Define the directory as a resource in Puppet:

```
file { "/etc/apt/sources.list.d":
    ensure  => directory,
    recurse => true,
    purge   => true,
}
```

The combination of `recurse` and `purge` will remove all files and subdirectories in `/etc/apt/sources.list.d` which are not deployed by Puppet. You can then deploy your own files to that location using a separate resource:

```
file { "/etc/apt/sources.list.d/bitfield.list":
    content => "deb http://packages.bitfieldconsulting.com/ lucid
main\n",
}
```

If there are subdirectories which contain files you don't want to purge, just define the subdirectory as a Puppet resource, and it will be left alone:

```
file { "/etc/exim4/conf.d/acl":
    ensure => directory,
}
```

Cleaning up old files

Puppet's `tidy` resource will help you clean up old or out-of-date files, reducing disk usage. For example, if you have Puppet reporting enabled as described in the section on generating reports, you might want to regularly delete old report files.

How to do it...

1. Add the following to your manifest:

   ```
   tidy { "/var/lib/puppet/reports":
       age     => "1w",
       recurse => true,
   }
   ```

2. Run Puppet:

   ```
   # puppet agent --test

   info: Retrieving plugin info: Caching catalog for cookbook.
   bitfieldconsulting.com

   notice: /Stage[main]//Node[cookbook]/Tidy[/var/lib/puppet/
   reports]: Tidying File[/var/lib/puppet/reports/cookbook.
   bitfieldconsulting.com/201102241546.yaml]

   notice: /Stage[main]//Node[cookbook]/Tidy[/var/lib/puppet/
   reports]: Tidying File[/var/lib/puppet/reports/cookbook.
   bitfieldconsulting.com/20110214727.yaml]

   …

   info: Applying configuration version '1306149187'

   notice: /File[/var/lib/puppet/reports/cookbook.bitfieldconsulting.
   com/201102241546.yaml]/ensure: removed

   notice: /File[/var/lib/puppet/reports/cookbook.bitfieldconsulting.
   com/201102141727.yaml]/ensure: removed …

   notice: Finished catalog run in 1.48 seconds
   ```

How it works...

Puppet searches the specified path for any files matching the `age` parameter: in this case, `1w` (one week). It also searches subdirectories (`recurse => true`).

Any files matching your criteria will be deleted.

There's more...

You can specify file ages in seconds, minutes, hours, days, or weeks by using a single character to specify the time unit, like this:

```
60s
180m
24h
30d
4w
```

You can specify that files greater than a given size should be removed, like this:

```
size => "100m",
```

removes files of 100 megabytes and over. For kilobytes, use `k`, and for bytes, use `b`.

Note that if you specify both `age` and `size` parameters, they are treated as independent criteria. For example, if you specify:

```
age  => "1d",
size => "512k",
```

Puppet will remove all files of 512KB or above, regardless of age, and all files older than one day, regardless of size.

Using schedules with resources

Using a `schedule` resource, you can control when other resources get applied. For example, the built-in `daily` schedule does what you'd expect: if you specify a resource like this:

```
exec { "/usr/bin/apt-get update":
    schedule => daily,
}
```

it'll be applied once a day.

The slightly tricky thing about `schedule` is that it doesn't guarantee that the resource will be applied once a day. It's just a limit: the resource won't be applied more than once a day. When or whether the resource is applied at all will depend on when and whether Puppet runs.

That being so, `schedule` is best used to restrict other resources: for example, you might want to make sure that `apt-get update` isn't run more than once an hour, or that a maintenance job doesn't run during daytime production hours.

For this you'll need to create your own `schedule` resources.

How to do it...

1. Add the following to your manifest:

   ```
   schedule { "not-in-office-hours":
       period => daily,
       range  => [ "17:00-23:59", "00:00-09:00" ],
       repeat => 1,
   }

   exec { "/bin/echo Doing maintenance!":
       schedule => "not-in-office-hours",
   }
   ```

2. Run Puppet.

How it works...

We've created a `schedule` called `not-in-office-hours` which specifies the repetition period as `daily`, and the allowable time range as after 5pm, or before 9am:

```
period => daily,
range  => [ "17:00-23:59", "00:00-09:00" ],
```

We've also said that the maximum number of times a resource can be applied in one period as 1:

```
    repeat => 1,
```

Now we apply that schedule to an `exec` resource:

```
exec { "/bin/echo Doing maintenance!":
    schedule => "not-in-office-hours",
}
```

Without the `schedule` parameter, this resource would be run every time Puppet runs. Now Puppet will check the `not-in-office-hours` schedule to see:

- whether the time is in the permitted range
- whether the resource has been run the maximum permitted number of times in this period

For example, let's consider what happens if Puppet runs every hour, on the hour:

- 4pm: it's outside the permitted time range, so Puppet will do nothing
- 5pm: it's inside the permitted time range, and the resource hasn't been run yet in this period, so Puppet will apply the resource
- 6pm: it's inside the permitted time range, but the resource has already been run once, so it's reached its maximum `repeat` count. Puppet will do nothing.

And so on until the next day.

There's more...

You can increase the `repeat` parameter if you want to, for example, run a job no more than 6 times an hour:

```
period => hourly,
repeat => 6,
```

Remember that this won't guarantee that the job is run 6 times an hour. It just sets an upper limit: no matter how often Puppet runs or anything else happens, the job won't be run if it has already run 6 times this hour. If Puppet only runs once a day, the job will just be run once. So `schedule` is best used for making sure things don't happen at certain times (or don't exceed a given frequency).

Auditing resources

I once had to diagnose a server which was failing to respond to `ping`, SSH, or console connections. The mystery was solved when I called the site where the machine was located. They informed me that two unidentified men had arrived earlier and simply carried the server out of the front door and into a truck. The message here is that it's good to know who's doing what to your servers.

Dry run mode, using the `--noop` switch, is a simple way to audit any changes to a machine under Puppet's control. However, Puppet also has a dedicated audit feature, which can report changes to resources or specific attributes.

How to do it...

1. Define a resource with the `audit` metaparameter:

```
file { "/etc/passwd":
    audit => [ owner, mode ],
}
```

How it works...

The `audit` metaparameter (a **metaparameter** is a parameter which can be applied to any resource, not just to specific types) tells Puppet that you want to record and monitor certain things about the resource. The value can be a list of the parameters which you want to audit.

In this case, when Puppet runs, it will now record the owner and mode of the `/etc/passwd` file. If either of these should change - for example, if you run:

```
# chmod 666 /etc/passwd
```

Puppet will pick up this change and log it on the next run:

```
notice: /Stage[main]//Node[cookbook]/File[/etc/passwd]/mode: audit
change: previously recorded value 644 has been changed to 666
```

There's more...

This feature is very useful for auditing large networks for any changes to machines, either malicious or accidental. You can use the `tagmail` reports feature to automatically send audit change notices by email. It's also very handy for keeping an eye on things which aren't managed by Puppet, for example application code on production servers. You can read more about Puppet's auditing capability here: `http://www.puppetlabs.com/blog/all-about-auditing-with-puppet/`

If you just want to audit everything about a resource, use `all`:

```
file { "/etc/passwd":
    audit => all,
}
```

See also

- Dry-running your Puppet manifests to avoid surprises
- E-mailing log messages containing specific tags

Temporarily disabling resources

Sometimes you want to disable a resource for the time being, so that it doesn't interfere with other work. For example, you might want to tweak a configuration file on the server until you have the exact settings you want, before checking it into Puppet. You don't want Puppet to overwrite it with an old version in the meantime, so you can set the `noop` metaparameter on the resource:

```
noop => true,
```

How to do it...

1. .Add the following to your manifest:

   ```
   file { "/tmp/test.cfg":
       content => "Hello, world!\n",
       noop => true,
   }
   ```

2. Run Puppet:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1306159566'
   notice: /Stage[main]//Node[cookbook]/File[/tmp/test.cfg]/ensure:
   is absent, should be file (noop)
   notice: Finished catalog run in 0.53 seconds
   ```

3. Now remove the `noop` parameter:

   ```
   file { "/tmp/test.cfg":
       content => "Hello, world!\n",
   }
   ```

4. Run Puppet again:

   ```
   # puppet agent --test
   info: Retrieving plugin
   info: Caching catalog for cookbook.bitfieldconsulting.com
   info: Applying configuration version '1306159705'
   notice: /Stage[main]//Node[cookbook]/File[/tmp/test.cfg]/ensure:
   defined content as '{md5}746308829575e17c3331bbcb00c0898b'
   notice: Finished catalog run in 0.52 seconds
   ```

How it works...

The first time we ran Puppet, the noop metaparameter was set to `true`, so for this particular resource it's as if you had run Puppet with the `--noop` flag. Puppet noted that the resource would have been applied, but otherwise did nothing.

In the second case, with `noop` removed, the resource is applied as normal.

Managing timezones

"I try to take one day at a time, but sometimes several days attack at once."

- Ashleigh Brilliant

Sooner or later, you'll encounter a weird problem which you'll eventually track down to servers having different time zones. It's wise to avoid this kind of issue by making sure all your servers use the same time zone, whatever their geographical location (GMT is the logical choice).

Unless a server is solar powered, I can't think of any reason for it to care about the time zone it's in.

How to do it...

1. Create the file `/etc/puppet/modules/admin/manifests/gmt.pp` with the following contents:

```
class admin::gmt {
    file { "/etc/localtime":
        ensure => "/usr/share/zoneinfo/GMT",
    }
}
```

2. Add the following to all nodes:

```
include admin::gmt
```

3. Run Puppet:

```
# puppet agent --test
info: Retrieving plugin
info: Caching catalog for cookbook.bitfieldconsulting.com
info: Applying configuration version '1304955158'
info: FileBucket adding /etc/localtime as {md5}02b73b0cf0d96e2f75c
ae56b178bf58e
info: /Stage[main]/Admin::Gmt/File[/etc/localtime]: Filebucketed
/etc/localtime to puppet with sum 02b73b0cf0d96e2f75cae56b178bf58e
notice: /Stage[main]/Admin::Gmt/File[/etc/localtime]/ensure:
ensure changed 'file' to 'link'
notice: Finished catalog run in 1.94 seconds
```

How it works...

No explanation necessary.

There's more...

If you want to use a different timezone, choose the appropriate file in `/usr/share/zoneinfo`: for example, `US/Eastern`.