Available as a
PDF Electronic Book
or Print On Demand

Pulling Strings with Puppet

Configuration Management Made Easy

192 PAGES

James Turnbull

Apress®
THE EXPERT'S VOICE™

Pulling Strings with Puppet

Configuration Management Made Easy

JAMES TURNBULL

firstPress™

Pulling Strings with Puppet: Configuration Management Made Easy

ISBN-13: 978-1-59059-978-5

ISBN-10: 1-59059-978-0

eISBN-13: 978-1-4302-0622-4

Printed and bound in the United States of America (POD)

Lead Editors: Jason Gilmore, Joseph Ottinger

Technical Reviewer: Dennis Matotek

Editorial Board: Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jason Gilmore, Kevin Goff, Jonathan Hassell, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Beth Christmas

Copy Editor: Ami Knox

Associate Production Director: Kari Brooks-Copony

Compositor: Richard Ables

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code/Download section.

This book is dedicated to Ruth Brown, who makes me laugh, and to my family for their continued support.

Contents

About the Author

JAMES TURNBULL works for the National Australia Bank as a Security Architect. He is the author of Hardening Linux, which focuses on hardening Linux hosts, and Pro Nagios 2.0, which focuses on enterprise management using the Nagios open source tool.

James has previously worked as an executive manager for IT security at the Commonwealth Bank of Australia, the CIO of a medical research foundation, manager of the architecture group of an outsourcing company, and in a number of IT roles in gaming, telecommunications, and government. He is an experienced infrastructure architect with a background in Linux/Unix, AS/400, Windows, and storage systems. He has been involved in security consulting, infrastructure security design, SLA, and service definition, and has an abiding interest in security metrics and measurement.

About the Technical Reviewer

DENNIS MATOTEK was born in a small town in Victoria, Australia called Mildura. Like all small towns, the chronic lack of good strong coffee drives the young to search further afield. Dennis moved to Melbourne where good strong coffee flows through the city in a river called the Yarra. However, it was in Scotland that Dennis was introduced to Systems Administration.

Scotland, on the technological edge, had 486DX PCs and a Vax. On arriving back in Melbourne, after staying awake for 24 hours at an airport minding his bags, Dennis was given a job interview—jobs in those days fell down like snowflakes from the sky.

Since that time, Dennis has stayed predominately in Melbourne working with IBM AS400s (iSeries) for 6 years and Linux for 7 years.

Dennis also wrote and directed some short films and plays. He has a lovely LP (life partner) and a new little boy called Zigfryd whom he misses terribly when at work, which is most of the time.

Acknowledgments

Luke Kanies—for writing Puppet and being kind enough to answer my numerous queries and questions.

The many members of the Puppet community who answered numerous questions and generally let me bother them.

Dennis Matotek for his technical review.

The team at Apress-Jason Gilmore, Joseph Ottinger, Beth Christmas, Ami Knox, Tina Nielsen, and Julie Miller-without all of you, none of this would be possible.

Jim Sumser for getting me started.

Introduction

This book introduces the reader to Puppet—a Ruby-based configuration management and automation tool for Linux and Unix platforms. The book is a beginning-to-intermediate guide to Puppet. It is aimed at system administrators, operators, systems engineers, and anyone else who has to manage Linux and Unix hosts.

This book requires a basic understanding of Linux/Unix systems administration including package management, user management, using a text editor such as vi, and some basic network and service management skills. If you wish to extend Puppet, you will need to have an understanding and some aptitude with the Ruby programming language. But for simple expansion of Puppet, basic Ruby skills are all that are needed. Additionally, as a programming language, Ruby is very approachable and easy to pick up.

The book starts with explaining how Puppet works and then moves on to installation and configuration. Each succeeding chapter introduces another facet of Puppet right up to demonstrating how you can extend Puppet yourself.

Chapter 1: Introduction to Puppet

Chapter 2: Installing and configuring Puppet

Chapter 3: Puppet's configuration language

Chapter 4: Using Puppet, which you learn through practical examples

Chapter 5: Reporting with Puppet

Chapter 6: Advanced Puppet features including integration with LDAP, performance management, and scalability

Chapter 7: Extending Puppet and Facter including adding your own Facter “facts” and Puppet configuration types

All of the source code, associated scripts, and configuration examples can be downloaded from the Apress web site. You can also submit any errata at the site.

If you have any questions and queries about the book, please do not hesitate to e-mail me at `james@hardening-linux.com`.

CHAPTER 1

Introducing Puppet

The lives of system administrators and in general individuals employed in IT's operational sector often revolve around a series of repetitive tasks: configuring hosts, creating users, and managing applications, daemons, and services. Often these tasks are repeated many times in the life cycle of one host in order to add new configuration or remedy configuration that has changed through error, entropy, or development. These tasks can be an ineffective use of time and effort.

The usual first response to these tasks is to try to automate them. This leads to the development of custom-built scripts and applications. In my first role as an administrator, I remember creating a collection of Control Language (CL) and Rexx scripts that I subsequently used to manage and operate a variety of infrastructure. Very little of the scripts developed in this ad hoc manner are ever published, documented, or reused. Indeed, copyright for most custom material rests with the operator or administrator's organization and is usually left behind when they move on. This leads to the same tool being developed over and over again.

Custom scripts and applications rarely scale to suit large environments and often have issues of stability, flexibility, and functionality. Such scripts also tend to suit only one target platform, resulting in situations such as the need to create a user creation script for BSD, one for Linux, and still another for Solaris. This increases the time and effort required to develop and maintain the very tools you are hoping to use to reduce administrative efforts.

Other approaches include the purchase of configuration management applications like Opsware, BMC's CONTROL-M, and CA's Unicenter products. But commercial tools generally suffer from two key issues: price and flexibility. Cost can quickly become an issue because the more types of platform and number of hosts that you are managing, the greater the cost. Commercial tools are also usually closed source and are limited to the features available to them, meaning that if you want to extend them, do something custom or specific to your environment, you need to request a new feature, potentially with a waiting period and associated cost.

Free and Open Source Software (FOSS) systems and configuration management tools offer an alternative to both custom and commercial solutions, offering two key opportunities for organizations:

- They are open and extensible.
- They are free!

With FOSS products, the tool's source code is at your fingertips, allowing you to develop your own enhancements or adjustments. You are also part of a community of developers who share the vision for the development of that tool. And you and your organization can in turn contribute to that vision. This ability to shape the direction of the tools you are using can certainly result in a more flexible outcome for your organization.

The price tag is also obviously an important consideration for the purchase of any tool. While you sacrifice paid support for many tools, you do get the tool itself at no cost. Additionally, the active Puppet community can and does provide support. With the price of many commercial configuration management tools running into hundreds of thousands of dollars, the potential cost savings from the use of an open source tool can be substantial.

In the configuration management space, a variety of open source tools are available, in various stages of development and maturity. Some of the key products in this area are

- *Puppet* (`http://puppet.reductivelabs.com/`):

  A configuration management tool written in Ruby with a client-server model that uses a declarative language to configure clients.

- *cfengine* (`http://www.cfengine.org/`):

  One of the first open source configuration management tools, released in 1993, it also has a client-server model and is commonly used in educational institutions.

- *LCFG* (`http://www.lcfg.org/`):

  A client-server configuration management tool that uses XML to define configuration.

- *Bcfg2* (`http://trac.mcs.anl.gov/projects/bcfg2`):

  A client-server configuration management tool written in Python. It uses specifications and the client responses to configure target hosts.

This book focuses on implementing and using one of these open source products, Puppet, to manage the configuration of your hosts, applications, daemons, and services.

What Is Puppet?

Puppet is an open source Ruby-based systems and configuration management tool relying upon a client-server deployment model. It is licensed using the GPLv2 license and is principally developed by Luke Kanies. Kanies has been involved in Unix and systems administration since 1997, and Puppet has developed from that experience. Unsatisfied with existing configuration management tools, Kanies began working in tool development in 2001 and in 2005 founded Reductive Labs, an open source development house focused on automation tools. Shortly after this, Reductive released their flagship product, Puppet.

What Makes Puppet Different?

Many systems and configuration management products, for example, cfengine, work in a similar manner. So what makes Puppet different? Puppet's defining characteristic is that it speaks the local language of your target hosts. This allows Puppet to define systems administration and configuration tasks with generic instructions on the Puppet server. These instructions are often called *recipes*.

Puppet's recipe syntax allows you to create a single script that allows you to create a user on all your target hosts. In turn, this recipe is interpreted and executed on each target host using the correct local syntax for that host. For instance, if the recipe is executed on a Red Hat Linux server, the user would be created with the `useradd` command. If the same recipe is executed on a FreeBSD target, the `adduser` command would be executed. Because Puppet recipes are so portable, community members and contributors share recipes for a variety of activities on the Puppet website, mailing list, and IRC channel!

The next area Puppet excels in is flexibility. As a result of its open source nature, you always have free access to Puppet's source code, meaning if you have a problem and you have the skills to do so, you can alter or enhance Puppet's code to suit your environment. Additionally, community developers and contributors regularly enhance and add to the functionality of Puppet. A large community of developers and users also contribute to providing documentation and support for Puppet.

Puppet is also readily extensible. Functions to support custom packages and configuration unique to your environment can be quickly and easily added to your Puppet installation. In addition, the Puppet community regularly adds code and packages that you can modify or incorporate into your environment.

I'll show you just how easy this is in Chapter 7. These additions can also be readily made without changing the core Puppet code, allowing you a clear upgrade and support path as Puppet develops.

Finally, Puppet makes use of another Ruby-based tool, Facter (`http://reductivelabs.com/projects/facter/`). Facter is a system analysis tool that allows you to query and return information about hosts that you can use in your Puppet configuration as variables. This means you can write generic configuration instructions and use Facter-returned variables to ensure the right values are configured on the right host. This precludes the need for external databases, configuration files, or directories. I'm going to look at Facter in more detail in Chapter 3 and how to add your own "facts" in Chapter 7.

How Does Puppet work?

With Puppet, central servers, called *Puppet masters*, are installed and configured. Client software is then installed on the target hosts, called *puppets* or *nodes*, that you wish to manage. Configuration is defined on the Puppet master, compiled, and then pushed out to the Puppet clients when they connect.

To provide the client-server connectivity, Puppet uses XML-RPC web services running over HTTPS on TCP port 8140. To provide security, the sessions are encrypted and authenticated with internally generated self-signed certificates. Each Puppet client generates a self-signed certificate that is then validated and authorized on the Puppet master.

Note ➡ Puppet currently uses XML-RPC web services, but at the time of writing a significant update and refactor of the code was taking place to migrate it to a REST-based web services model (`http://en.wikipedia.org/wiki/Representational_State_Transfer`). This migration should provide much more efficient and elegant web service functionality.

Thereafter each client contacts the server, by default every half hour, to confirm that its configuration is up to date. If new configuration is available or the configuration has changed, it is recompiled and then applied to the client. If required, a configuration update can also be triggered from the server, forcing configuration down to the client. If any existing configuration has varied on the client, it is corrected with the original configuration from the server. The results of any activity are logged and transmitted to the server.

You can see Puppet's client server model in Figure 1-1.

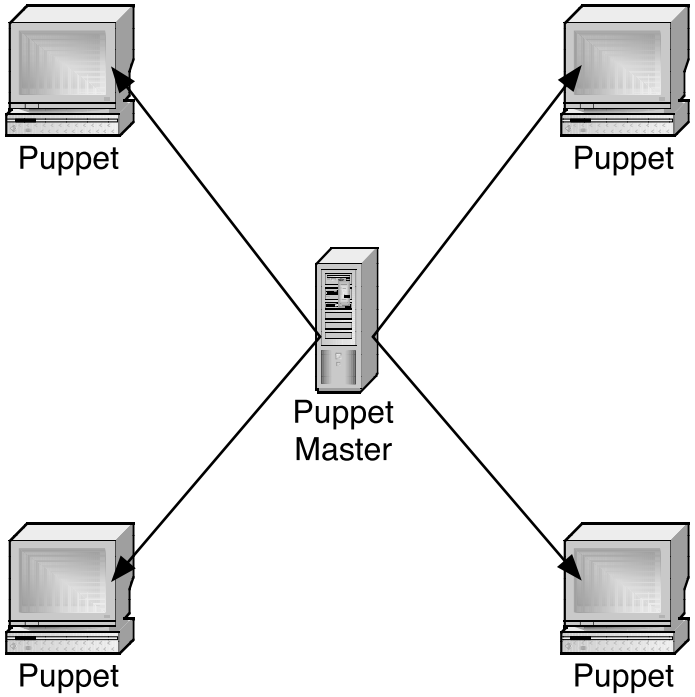

Figure 1-1. Puppet client-server model

Puppet, however, is more than just a client-server configuration management tool—it's a tripartite architecture combining a declarative language, transactional layer, and resource abstraction layer. Let's take a closer look at each.

A Declarative Language

At the heart of how Puppet works is a language that allows you to articulate and express your configuration. Your configuration components are organized into entities called *resources*, which in turn can be grouped together in *collections*. Resources are made up of a *type*, *title*, and a series of *attributes*. You can see an example of a simple resource in Listing 1-1.

Listing 1-1. A Puppet Resource

```
file { "/etc/passwd":
       owner => "root"
}
```

The resource in Listing 1-1 alters the configuration of the `/etc/passwd` file, changing its ownership to the `root` user. Inside Listing 1-1 the type being used is the `file` type. The resource type tells Puppet what type of resource you are managing, for example, the user and file types that are used for managing user and file operations on your nodes, respectively. Puppet comes with a number of resource types by default including types to manage files, services, packages, cron jobs, and file systems, among others. Inside our type we've specified the file to be managed, `/etc/passwd`; this becomes the title of our resource. When referring to the resource in other parts of our configuration, we'd reference this title. Lastly, we've specified a single attribute, `owner`, which tells Puppet to set the ownership of the file to the root user.

Note ➡ As you'll learn in Chapter 7, you can also extend Puppet to add your own resource types.

We can extend beyond these single resources with collections. Collections allow you to group together many resources; for example, an application such as Apache is made up of a package, a service, and a number of configuration files. Puppet calls these collections *classes*. Each of these components would be represented as a resource (or resources) and then collected together in a class and applied to a node.

The Puppet language also defines the nodes you wish to configure. After a client is connected to Puppet, a node definition can be created that defines what resources and collections of resources are applied to each node. This allows you to apply appropriate configuration to all nodes running a particular platform or a particular service, for example, specifying all resources required for Red Hat Enterprise Linux nodes or all configuration required for a database or web server.

The Puppet language also allows the use of features you'll find in many programming languages, such as variables, arrays, and conditional statements and clauses. We'll examine Puppet's language in detail in Chapter 3.

A Transactional Layer

Puppet's transactional layer provides the engine of the Puppet client-server deployment. Configurations are created and can be executed repeatedly on the target hosts. The Puppet application architecture describes this sort of configuration as *idempotent*, meaning multiple applications of the same operation will yield the same results.

Puppet is not fully transactional; your transactions aren't logged (other than informative logging) and hence you can't roll back transactions as you can with many databases. You can, however, model transactions in a noop (no operation) mode that allows you to test the execution of your changes without actually making any changes.

A Resource Abstraction Layer

Lastly, Puppet provides an abstraction layer between your platform and the description of your configuration. The resources defined in Puppet to configure your nodes are independent from the commands, formats, and syntax required to configure those resources locally on your nodes. So it does not matter whether you want to create a user on one of many platforms, Puppet considers the definition of that user to be identical.

Puppet does this abstraction using *providers*. Providers are implementations of resource types. In the provider model, a resource is defined in Puppet. The resource is then set to be applied on a node or nodes. Puppet then detects the platform of the node, and the appropriate provider for that platform is then called and used to actually implement the configuration on the node. For example, when creating a user, we define a user type resource in Puppet. We then tell Puppet that we want to create that user on all Mac OS X and OpenBSD nodes. Then when an OS X node connects, the OS X user provider is called and the user created. When an OpenBSD node connects, the OpenBSD user provider is called and the user created. Some platforms share providers, for example, managing files on many platforms is similar, if not identical. Some other resource types, for example, the package resource type, which installs and manages software packages, have numerous providers, as package management is different on a variety of platforms.

Puppet Performance and Hardware

Understanding of Puppet's scalability and performance is still immature at the time of writing. There are two facets to performance management—the number of nodes connected

and the amount of configuration defined on each node. There are no clear-cut guidelines around how many nodes and how much configuration on each can be supported on a single master server or around the scale and capacity of hardware required to run the master. Anecdotal evidence suggests that 50 to 100 nodes with a moderate amount of configuration can be managed on a single CPU master with 2GBs of RAM. More nodes will obviously require scaled up hardware.

Internally Puppet uses the WEBrick web server to interface with clients. The WEBrick server does have performance limitations and hence does not provide a fully scalable solution. As an alternative, Puppet also has the capability of internally making use of the more scalable Mongrel web server instead of WEBrick. A load balancer, such as Apache with `mod_proxy` or Pound, is then placed in front of Puppet. This allows the use of multiple load-balanced Puppet master instances, which should result in a more scalable solution. Generally, the WEBrick web server no longer performs adequately if you are managing 50 or more nodes, and migration to Mongrel will probably be needed.

Note ➡ In Chapter 6, I'll demonstrate how to replace WEBrick with Mongrel.

Sites have reported that when running Puppet with load balancing and Mongrel, node volumes of 5000 or more are feasible with appropriate hardware.

The Future for Puppet

Lastly, it is very important to remember that Puppet is a young tool and is still in the midst of development and change. The Puppet community is growing quickly, and many new ideas, developments, patches, and recipes appear every day. But this does make it important to keep an eye on the Puppet mailing lists and the IRC channel, #puppet on Freenode, as new enhancements that could help you better manage your configurations appear frequently.

Resources

There are a number of useful resources available to get you started with Puppet.

Web

- *Puppet documentation*:
  `http://reductivelabs.com/trac/puppet/wiki/DocumentationStart`
- *Puppet FAQ*:
  `http://reductivelabs.com/trac/puppet/wiki/FrequentlyAskedQuestions`
- *About Puppet*:
  `http://reductivelabs.com/trac/puppet/wiki/AboutPuppet`
- *Introduction to Puppet*:
  `http://reductivelabs.com/trac/puppet/wiki/PuppetIntroduction`
- *Who is using Puppet*:
  `http://reductivelabs.com/trac/puppet/wiki/WhosUsingPuppet`

Mailing Lists

- *Puppet mailing lists*:
  `http://reductivelabs.com/trac/puppet/wiki/GettingHelp`

IRC

- *Puppet IRC channel*:
  `irc://irc.freenode.net/puppet`

CHAPTER 2

Installing and Running Puppet

This chapter focuses on installing and running Puppet master servers and clients (also known as nodes). There are a variety of methods you can use to install Puppet masters and clients: from source, packages, or as a Ruby Gem. This chapter will take you through the steps required for installation using each of these methods.

The Puppet server and clients are designed to run on Unix and Linux platforms; currently there is no port for Windows (although it may be possible to run Puppet under Cygwin). This book will only cover installation on Unix and Linux platforms. Both the master server and the clients will run successfully on a variety of BSD flavors, Linux distributions, Sun Solaris, Mac OS X, and indeed most Unix-like platforms that support Ruby.

The chapter will also take you through configuring and running both the Puppet master and client. By the end of this chapter, you should have an introduction to how both the master and clients can be configured. You will also be able to start and stop the master and clients on a variety of platforms.

Note ➡ When referring to the Puppet client, we'll distinguish between the terms *client* and *node*. The term *client* refers to the Puppet client daemon that connects to the Puppet master and retrieves the configuration. The term *node* refers to the underlying host to which configuration is applied.

Installation Prerequisites

The process of installing Puppet's master and client components is quick and easy, but you will need to install some prerequisites first. The prerequisites are required for both hosts that run the Puppet master or client (or both—your Puppet master can also be a Puppet node). These prerequisites include the Ruby interpreter, select Ruby libraries, and Facter.

Installing Ruby

As Puppet is a Ruby-based application, the first thing you need to ensure you have installed is Ruby and a few key Ruby libraries. These days, many Linux and other Unix-like platforms come with a Ruby package, and you can install this package and any required library packages. If your distribution does not have a package, you can install Ruby from source.

Installing Ruby from Source

You can download the latest Ruby source package from `http://www.ruby-lang.org/en/downloads/`. To support Puppet, you will need at least Ruby version 1.8.1 or higher. The current Ruby release, at the time of writing, is 1.8.6. Download the source package and unpack it.

```
# wget ftp://ftp.ruby-lang.org/pub/ruby/ruby-1.8.6.tar.gz
# tar -zxf ruby-1.8.6.tar.gz
```

Change into the resulting directory and configure the package.

```
# cd ruby-1.8.6
# ./configure
```

By default, on most systems the Ruby binary will be installed into `/usr/local/bin`. You can override this with the `--prefix` configure option like so:

```
# ./configure --prefix=/usr
```

Here I've specified `/usr` as the target prefix directory. Next, we need to make and install the Ruby files like so:

```
# make
# make install
```

You can now test that Ruby is installed by executing the following:

```
# ruby --version
ruby 1.8.6 (2007-03-13 patchlevel 0) [i686-linux]
```

If your Ruby version is returned, installation has been successful. If it is not returned, confirm the `ruby` binary is in your path or check any error messages from the `make` and/or installation process. Installing Ruby in this manner will install all of the required libraries that Puppet needs to function properly.

Installing Ruby and Ruby Libraries from Packages

Many Linux distributions and Unix operating systems have Ruby packages available for them. These include Red Hat Enterprise Linux and Fedora, Debian, Ubuntu, SuSE, and Mandriva. Some distributions bundle all the required Ruby binaries and libraries in a single package. Other distributions separate the core development environment and the libraries into individual packages. In Table 2-1, I have detailed the package and/or port names for the required packages for a variety of BSD and Linux distributions.

Table 2-1. Package Names for Ruby and Ruby Libraries

OS	Ruby	Ruby Libraries	Additional Package
Debian	`ruby`	`librubylibopenssl-ruby`	`libxmlrpc-ruby`
FreeBSD	`ruby`		
Gentoo	`ruby`		
Mandriva	`ruby`		
NetBSD	`ruby`		
OpenBSD	`ruby`		
Red Hat	`ruby`	`ruby-libs`	
SuSE	`ruby`		
Ubuntu	`ruby`	`librubylibopenssl-ruby`	`libxmlrpc-ruby`

So, if we're installing Ruby and its libraries on a Red Hat Fedora host, we need to use its package management system to install the `ruby` and `ruby-libs` packages like so:

```
# yum install ruby ruby-libs
```

Installing the Ruby package and libraries may not always install all of the required libraries. If the following base libraries are not installed as part of your base Ruby installation, you may need to selectively install the missing libraries.