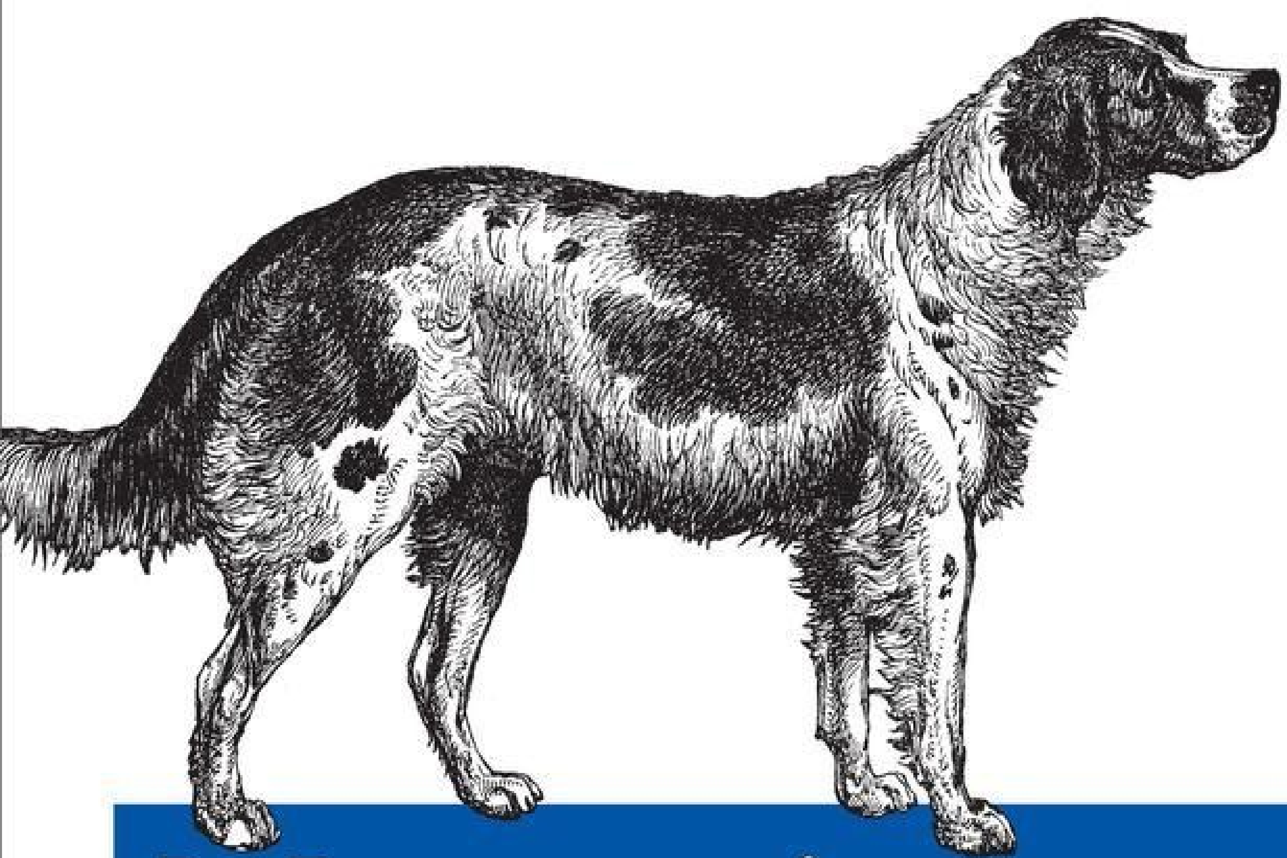
Configuration Management at Scale



Managing Infrastructure with Puppet

O'REILLY®

James Loope

Managing Infrastructure with Puppet

Table of Contents

Managing Infrastructure with Puppet

James Loope

Editor

Mike Loukides

Editor

Meghan Blanchette

O'Reilly Media

Preface

This book is for anyone using or considering Puppet as a systems automation tool. Readers of this book should be familiar with Linux systems administration and basic Ruby. I’ll cover the basics of using Puppet manifests for configuration management and techniques for executing and managing those configurations with MCollective and Facter. I’ll often make suggestions that assume you are managing a virtualized infrastructure, but virtualization is not necessary to reap the benefits of this software.

Software

This book is focused on Puppet 2.6.1 with Facter 1.5.6, and the MCollective version used is 1.0.1. Because of the very active development of all of these products, concepts and examples may not apply to earlier versions.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

Tip

This icon signifies a tip, suggestion, or general note.

Caution

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Managing Infrastructure with Puppet* by James Loope (O'Reilly). Copyright 2011 James Loope, 978-1-449-30763-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/0636920020875/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Chapter 1. Baby Steps to Automation

Puppet is a configuration management framework with an object-oriented twist. It provides a declarative language syntax and an abstraction layer that allow you to write heavily reusable and understandable configuration definitions. In this chapter, I’ll cover the basics of the Puppet programs, the language syntax, and some simple class and resource definitions.

Getting the Software

A Puppet deployment comes with a couple of pieces of software. For the most part, these can be installed from your chosen Linux distribution's package manager. Alternatively, you can use the packages or source provided by Puppet Labs at http://www.puppetlabs.com/misc/download-options/. In my examples, I've used Ubuntu Linux 11.04, but the packages are very similar in each distro. There are generally two packages: the Puppet package itself, which comes with Facter, and the Puppet Master server. For the purposes of this chapter, the Puppet and Facter package will suffice. When installed, it will include an init script to start an "agent" daemon at boot, which will look for a Puppet Master. For simplicity's sake, we will test manifests from the command line using the `puppet apply` command to begin:

- Ubuntu: `apt-get install puppet`
- Fedora: `yum install puppet`
- Mac OS X: `port install puppet`

Introducing Puppet

Puppet helps you organize and execute configuration plans on servers. This is enabled through a resource abstraction layer that allows you to address the different configurable components of your system as generic objects. In the Puppet view, a server is a collection of resource objects that have a set of particular attributes that describe how that object looks.

It is your job to build a catalog of resource declarations that will tell Puppet how those resources should look when properly configured. When Puppet implements a catalog, it compares the existing resources on the server to the ones that you have defined in your descriptions. It then decides on a set of changes that need to occur to bring the catalog state into agreement with your descriptions. The execution is *idempotent*, meaning that only the changes needed to bring the state into agreement with the description will be made. The entire catalog can be run over and over again without causing deviation from the described state.

These resource descriptions are made in a Domain Specific Language implemented in Ruby. This means that the syntax is often similar to Ruby, but you cannot simply write Ruby code in a Puppet manifest and have it executed. In fact, the language is declarative, rather than imperative like Ruby. With Puppet, you say how you want things to look, as opposed to describing what should be done to make them look that way. It’s Puppet’s job to know how to make that description reality.

Putting the Pieces Together

So Puppet lets us describe our server configurations and then goes off and does all of the work for us. But how does that happen? There are a couple different ways that Puppet can manage your systems, depending on your scale and needs.

Puppet

The first piece is the Puppet program itself. It’s an executable Ruby program that has the majority of Puppet’s functionality rolled up and made accessible via the command line. With the Puppet program, you can syntax check your Puppet code, apply the resources to a machine manually, describe the current state of the world as seen by the abstraction layer, and get some documentation of Puppet’s workings.

Puppet Master

When we need to apply our Puppet configurations to a large number of servers, it becomes laborious to log in to each machine, copy our configurations to it, and execute the Puppet command against them. We are better served by keeping all of our configurations in a central location, defining which configurations apply to which servers, and then letting Puppet do the work of pulling the configurations from the repository and applying them. To enable this client-server behavior, Puppet has a network daemon called the Puppet Master.

The Puppet program can be run in a daemonized mode by the server init and is then referred to as a Puppet agent. The agents talk to the Puppet Master over client-certificate authenticated SSL and the master hands out their configuration catalog. In its default configuration, the agents work in a polling mode and check in for catalog updates every 30 minutes. This allows us to store our configurations in a central location without having to worry about keeping all of our systems catalogs in sync through some out-of-band means.

Getting Started

Once Puppet is installed, you will have the `puppet` command at your disposal. The first thing you should do is run **`puppet describe --list`**. This will provide a list of the available resource “types” you have to work with out of the box:

```
:> puppet describe --list
    These are the types known to puppet:
augeas           - Apply the changes (single or array of changes ...
computer         - Computer object management using DirectorySer ...
cron             - Installs and manages cron jobs
exec             - Executes external commands
file             - Manages local files, including setting owners ...
filebucket       - A repository for backing up files
group            - Manage groups
host             - Installs and manages host entries
k5login          - Manage the `
macauthorization - Manage the Mac OS X authorization database
mailalias        - Creates an email alias in the local alias dat ...
maillist         - Manage email lists
mcx              - MCX object management using DirectoryService  ...
mount            - Manages mounted filesystems, including puttin ...
nagios_command   - The Nagios type command
nagios_contact   - The Nagios type contact
nagios_contactgroup - The Nagios type contactgroup
nagios_host      - The Nagios type host
nagios_hostdependency - The Nagios type hostdependency
nagios_hostescalation - The Nagios type hostescalation
nagios_hostextinfo - The Nagios type hostextinfo
nagios_hostgroup - The Nagios type hostgroup
nagios_service   - The Nagios type service
nagios_servicedependency - The Nagios type servicedependency
nagios_serviceescalation - The Nagios type serviceescalation
nagios_serviceextinfo - The Nagios type serviceextinfo
nagios_servicegroup - The Nagios type servicegroup
nagios_timeperiod - The Nagios type timeperiod
notify           - Sends an arbitrary message to the agent run-t ...
package          - Manage packages
resources        - This is a metatype that can manage other reso ...
schedule         - Defined schedules for Puppet
selboolean       - Manages SELinux booleans on systems with SELi ...
selmodule        - Manages loading and unloading of SELinux poli ...
service          - Manage running services
ssh_authorized_key - Manages SSH authorized keys
sshkey           - Installs and manages ssh host keys
stage            - A resource type for specifying run stages
tidy             - Remove unwanted files based on specific crite ...
user             - Manage users
whit             - The smallest possible resource type, for when ...
yumrepo          - The client-side description of a yum reposito ...
zfs              - Manage zfs
zone             - Solaris zones
zpool            - Manage zpools
```

We’ll primarily be concerned with the file, exec, cron, user, group, and package types. In addition to these built-in types, a large variety of user-contributed modules add functionality for nearly every commonly used configuration scenario. Documentation of the built-in types can be found on the Puppet Labs documentation site at http://docs.puppetlabs.com/references/2.6.0/type.html.

To get some detail about each of these resource types, you can use `puppet describe` **`type`**. This will output Puppet’s documentation on that particular resource type including parameters and often usage examples as well:

```
:> puppet describe host

host
====
Installs and manages host entries.  For most systems, these
entries will just be in `/etc/hosts`, but some systems (notably OS X)
will have different solutions.


Parameters
----------

- **ensure**
    The basic property that the resource should be in.  Valid values are
    `present`, `absent`.

- **host_aliases**
    Any aliases the host might have.  Multiple values must be
    specified as an array.

- **ip**
    The host's IP address, IPv4 or IPv6.

- **name**
    The host name.

- **target**
    The file in which to store service information.  Only used by
    those providers that write to disk.

Providers
---------
parsed
```

Note

`puppet describe **type** -s` will give you a less verbose description. This is useful if you just want to know the correct name of a parameter without having to grep through pages of text.

You can also use Puppet to make queries to the resource abstraction layer and return the current state of things on a system. This makes reproducing a particular configuration on an existing system easy when there is a supported resource type. The command for this is `puppet resource` **`type name`**. Here is an example query using the host resource:

```
:> puppet resource host

host { 'example.example.com':
    host_aliases => ['example'],
    target => '/etc/hosts',
    ip => '10.0.1.101',
    ensure => 'present'
}
host { 'localhost':
    target => '/etc/hosts',
    ip => '127.0.0.1',
    ensure => 'present'
}

:> puppet resource host example.example.com

host { 'example.example.com':
    host_aliases => ['example'],
    target => '/etc/hosts',
    ip => '10.0.1.101',
    ensure => 'present'
}
```

Resource types are the building blocks of Puppet configurations and most of your time will be spent using them or writing new types to suit your needs. Let’s start with a simple declaration of a package resource.

Files and Packages

This first statement declares that the package `ntp` should be installed and that the file *ntp.conf* should be defined with the given contents and permissions at the path */etc/ntp.conf*, but only after the package `ntp` is installed. You can go ahead and test this out (on a test system!) by saving the above text to *test.pp* and executing **`puppet apply test.pp`**. When this manifest is run against a blank system, the agent will check for the existence of an `ntp` package and install it if necessary. Then the file at */etc/ntp.conf* will be installed if it doesn’t exist or overwritten with the content specified if it differs:

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

file { 'ntp.conf':
    path => '/etc/ntp.conf',
    mode => 640
    content => '
    driftfile /var/lib/ntp/ntp.drift
    statistics loopstats peerstats clockstats
    filegen loopstats file loopstats type day enable
    filegen peerstats file peerstats type day enable
    filegen clockstats file clockstats type day enable
    server 0.pool.ntp.org
    server 1.pool.ntp.org
    restrict -4 default kod notrap nomodify nopeer noquery
    restrict -6 default kod notrap nomodify nopeer noquery
    restrict 127.0.0.1
    restrict ::1
    ',
    require => Package[ntp],
}
```

A few notes here about the syntax: The capitalization of type in resources is important. You can see that when the resources file and package are declared, they are not capitalized, but when the file resource references the `ntp` package, it is capitalized. Always capitalize the first letter in the type when you are referring to a resource that you have declared elsewhere, but do not capitalize the type in the declaration itself. Also notice that the package declaration at the top is a sort of shortened form, leaving out line breaks and the comma at the end of the single parameter. The last comma is optional on a parameter list, but it is generally included in the full form.

The path, mode, and content parameters are fairly mundane, but the require parameter is special magic. The Puppet agent doesn’t have any innate sense of order of execution when it is run on a manifest or set of manifests. Things will happen in random sequence unless constrained by some dependencies. `require` is one of those dependencies. The above statement specifies that the file definition *ntp.conf* requires that the package `ntp` be installed before it is created. Conversely, we could have specified in the package declaration for `ntp` that it be run `before => File['ntp.conf']`. Next, we’ll look at a slightly more streamlined implementation:

```
package { 'ntp': ensure => '1:4.2.6.p2+dfsg-1ubuntu5' }

file { '/etc/ntp.conf':
        mode => '640',
        owner => root,
        group => root,
        source => '/mnt/nfs/configs/ntp.conf',
        require => Package[ntp],
    }
```

The most obvious change here is that we’ve moved the file content to an external source. We’ve told Puppet to go and look in */etc/nfs/configs* for a file named *ntp.conf* and put it in */etc/ntp.conf*. For the moment, we’ll use an NFS mount to distribute our configuration files. In later examples, we can use Puppet’s built-in artifice for that purpose. It’s good practice to specify both file permissions and ownership in your manifests, as well as package versions. I’ve replaced the ensure value with an explicit `ntp` package version. Puppet is intended to be used to make configuration changes as well as to ensure the correctness of configurations. You can think of it both as a deployment script and an auditing tool; by being explicit with your definitions, you can be very confident that your deployment will always work the same way. Finally, I’ll note that this file resource lacks an explicit path parameter. This is because, in Puppet, each type has a parameter that defaults to the resource name. This is referred to as the `namevar`, and for the `file` type, it is the `source`.

Services and Subscriptions

Let's add a watchdog to ensure that the `ntp` daemon that we've installed is actually running. This will give us some insurance that the proper services have been started, but by no means should it be considered a replacement for a service manager daemon.

I've added a service definition that `subscribes` to the `ntp` package and its configuration file. On execution, this definition will look in the process table for the pattern "ntpd". If it fails to find a match for the pattern, Puppet will start the `ntp` service to ensure that it is running. It also holds a subscription to the `ntp` package and the file at */etc/ntp.conf*. If we later change the config file or update the package version, Puppet will restart the service automatically:

```
package { 'ntp': ensure => '1:4.2.6.p2+dfsg-1ubuntu5' }

file { '/etc/ntp.conf':
        mode => 640
        owner => root,
        group => root,
        source => '/mnt/nfs/configs/ntp.conf',
        require => Package[ntp],
        }

service { "ntp":
    ensure => running,
    enable => true,
    pattern => 'ntpd',
    subscribe => [Package["ntp"], File["/etc/ntp.conf"]],
}
```

Warning

Make sure to test the behavior of the service you are managing. It may be innocuous to restart `ntp` when the config changes, but it's an ugly mess when you push a change that, unforeseen, restarts your production database.

Exec and Notify

Subscribing a service to a file is very convenient, but what if we need to do something more explicit when a file resource changes? I'll use a postfix transport map as an example. When this file is updated, I want to run postmap to compile the *transport.db* file.

In this example, I've specified an exec resource. This is the "brute force" resource in Puppet. You can use it to execute commands and shell scripts of your choosing, but there is an important caveat. The command must be idempotent. This means that your system configuration must be able to cope with having the command run over and over again. An exec type resource will generally be run on every Puppet run. The following example specifies that the command should not run unless the subscription to the */etc/transport* file is changed and a *refresh* is triggered. This is accomplished with the `refreshonly` parameter. Any exec can be refreshed either by a subscription or a notification. Notification works in the reverse of a subscription:

```
file { "/etc/postfix/transport":
        mode => 640
        owner => root,
        group => postfix,
        source => '/mnt/postfix/configs/transport',
        }
exec { "postmap /etc/postfix/transport":
        subscribe => File["/etc/postfix/transport"],
        refreshonly => true,
        }
```

Here we have the file resource notifying the exec of a change. Note that *notify* implies the behavior that would be seen with a before parameter and *subscribe* implies the ordering of a require parameter. In this example, the file will be created before the exec is run, and in the former example, the exec requires that the file be run first:

```
file { "/etc/postfix/transport":
        mode => 640
        owner => root,
        group => postfix,
        source => '/mnt/postfix/configs/transport',
        notify => Exec["postmap /etc/postfix/transport"],
        }
exec { "postmap /etc/postfix/transport":
        refreshonly => true,
        }
```

There are a couple of scenarios where you might want to use an exec, but only when some other condition requires it. Exec can be used to generate a file; for example, if I wish to fetch a configuration file that I've published on a web server.

In the first example, Puppet understands that the result of the exec is to create the file listed in the `creates` parameter. This exec will only be run if that file doesn't exist. The second example has the same effect, but it does so using a more customizable condition. The command will only be run if the exit status of the command in the `onlyif` parameter is zero. Nonzero status will cause the exec to be skipped:

```
exec { 'curl http://example.com/config/my.conf -o "/etc/myapp/my.conf"':
    creates => "/etc/myapp/my.conf",
    }
```