Puppet Documentation | Puppet Labs

June 30, 2011

Contents

Chapter 1

Puppet Labs Documentation

Welcome to the Puppet Labs documentation site. The documentation posted here is also available as a (very large, and frequently updated) PDF, which can be found here.

MCollective

For information about MCollective, see the Marionette Collective documentation.

Puppet Dashboard

For information about Puppet Dashboard, see the Puppet Dashboard documentation.

Drive-Thru

Small documents for getting help fast.

-  Core Types Cheat Sheet — available in single-page flavor (double-sided), extra breathing room flavor (six pages), and plain web page flavor
- Frequently Asked Questions

Learning Puppet

Learn to use Puppet! New users: start here.

- Introduction and Index
- Resources and the RAL — learn about resources, the molecules of system configuration
- Manifests — start writing and applying Puppet code
- Ordering — learn to join resources that depend on each other

- Variables, Facts, and Conditionals — read system information to make versatile manifests
- Classes and Modules, Part One — start collecting resources into self-contained modules

---

Reference Shelf

Get detailed information about config files, APIs, and the Puppet language.

- REST API — reference of api accessible resources
- Puppet Language Guide — all the language details
- Puppet Manpages — detailed help for each Puppet application

Generated References

Complete and up-to-date references for Puppet's resource types, functions, metaparameters, configuration options, indirection termini, and reports, served piping hot directly from the source code.

- Resource Types — all default types
- Functions — all built in functions
- Metaparameters — all type-independent resource attributes
- Configuration — all configuration file settings
- Report — all available report handlers

These references are automatically generated from the inline documentation in Puppet's source code. References generated from each version of Puppet are archived here:

- Versioned References — inline reference docs from Puppet's past and present

---

Puppet Guides

Learn about different areas of Puppet, fix problems, and design solutions.

Components

Learn more about major working parts of the Puppet system.

- Puppet commands: master, agent, apply, resource, and more — components of the system

Installing and Configuring

Get Puppet up and running at your site.

- An Introduction to Puppet
- Supported Platforms
- Installing Puppet — from packages, source, or gems
- Setting Up Puppet — includes server setup & testing
- Configuring Puppet — use `puppet.conf` to configure Puppet's behavior

Tuning and Scaling

Puppet's default configuration is meant for prototyping and designing a site. Once you're ready for production deployment, learn how to adjust Puppet for peak performance.

- Scaling Puppet — general tips & tricks
- Scaling With Passenger — for Puppet 0.24.6 and later
- Scaling With Mongrel — for older versions of Puppet

Basic Features and Use

- Techniques — common design patterns, tips, and tricks
- Troubleshooting — avoid common problems and confusions
- Puppet Modules — modules make it easy to organize and share content
- Parameterized Classes — use parameterized classes to write more effective, versatile, and encapsulated code
- Module Smoke Testing — write and run basic smoke tests for your modules
- Scope and Puppet — understand and banish dynamic lookup warnings with Puppet 2.7
- Puppet File Serving — serving files with Puppet
- Style Guide — Puppet community conventions
- Best Practices — use Puppet effectively

Advanced Features

Go beyond basic manifests.

- Templating — template out config files using ERB
- Virtual Resources
- Exported Resources — share data between hosts
- Environments — separate dev, stage, & production
- Reporting — learn what your nodes are up to
- Getting Started With CloudPack — create and bootstrap new nodes with the experimental CloudPack extension

Hacking and Extending

Build your own tools and workflows on top of Puppet.

Using APIs and Interfaces

- REST Access Control — secure API access with auth.conf
- External Nodes — specify what your machines do using external data sources
- Inventory Service — use Puppet's inventory of nodes at your site in your own custom applications

Using Ruby Plugins

- Plugins In Modules — where to put plugins, how to sync to clients
- Writing Custom Facts
- Writing Custom Functions
- Writing Custom Types & Providers
- Complete Resource Example — more information on custom types & providers
- Provider Development — more about providers

Developing Puppet

- Running Puppet from Source — preview the leading edge
- Development Life Cycle — learn how to contribute code
- Puppet Internals — understand how Puppet works internally

Other Resources

- Puppet Wiki & Bug Tracker
- Puppet Patterns (Recipes)

Help Improve This Document

This document belongs to the community and is licensed under the Creative Commons. You can help improve it!

To contribute ideas, problems, or suggestions, simply use the Contribute link. If you would like to submit your own content, the process is easy. You can fork the project on github, make changes, and send us a pull request. See the README files in the project for more information.

Documentation Version

This release of the documentation was generated from revision 1eeac62f293fcf764c4c32e1c9c27b08904248d5 of the puppet-docs repo on now.

Chapter 2

Learning Puppet

The web (including this site) is full of guides for how to solve specific problems with Puppet and how to get Puppet running. This is something slightly different.

Start: Resources and the RAL →

Latest: Modules (Part One) →

Welcome

This is **Learning Puppet,** and it's part of the Puppet documentation. Specifically, it's the first part.

By which I don't mean it's about getting Puppet installed, or making sure your SSL certificates got issued correctly; that's the *other* first part. To be a little gnomic about it — because why not — this series is less about how to use Puppet than it is about how to become a Puppet user. If you've heard good things about Puppet but don't know where to start, this, hopefully, is it.

It's a work in progress, and I'd love to read your feedback at nick.fagerlund@puppetlabs.com.

Get Equipped

You can't make a knowledge omelette without breaking… stuff. Possibly eggs, maybe your system's entire configuration! Such is life.

So to learn Puppet effectively, you need a virtual machine you can experiment on fearlessly. And to learn Puppet *fast,* you want a virtual machine with Puppet already installed, so you don't have to learn to debug SSL problems before you know how to classify a node.

In short, you want *this* virtual machine:

Get the Learning Puppet VM

The root user's password is `puppet`, and for your convenience, the system is configured to write its current IP address to the login screen about ten seconds after it boots.

If you'd rather cook up your own VM than download one from the web, you can imitate it fairly easily: this is a stripped-down CentOS 5.5 system with a hostname of "learn.puppet.demo," Puppet Enterprise installed using all default answers, iptables turned off, and the `pe-puppet` and `pe-httpd` services stopped and disabled. (It also has Puppet language modes installed for Vim and Emacs, but that's not strictly necessary.)

To begin with, you won't need separate agent and master VMs; you'll be running Puppet in its serverless mode on a single machine. When we get to agent/master Puppet, we'll walk through turning on the puppet master and duplicating this system into a new agent node.

Compatibility Notes

The Learning Puppet VM is available in VMWare .vmx format and the cross-platform OVF format, and has been tested with VMWare Fusion and VirtualBox.

Getting the VM working with VMWare is fairly simple, but some extra effort is necessary on VirtualBox — the IP address it prints isn't externally reachable, and by default you'll be unable to SSH to the VM. You can enable SSH by turning on port forwarding, but since several examples (and the eventual agent/master exercises) will require more network access than simply port 22, it's wiser to configure two network interfaces:

- Before starting the VM, modify its network settings to add "Host Only" network access on adapter 2.
- Next, run `ifconfig` on your host machine and confirm the subnet assigned to the `vboxnet0` interface. (By default, this is 192.168.56.x, and your host machine's IP address is 192.168.56.1.)
- Once the VM is running, log in on its console and run `ifconfig eth1 192.168.56.2` (or some other IP address compatible with the relevant subnet); this should let you ping and SSH the box from your host machine, and you can add an entry to your host machine's `/etc/hosts` file to make things more convenient.

Beyond this, teaching the use of virtualization software is outside the scope of this introduction, but let me know if you run into trouble, and we'll try to refine our approach over time.

Hit the Gas

And with that, you're ready to start.

Part one: Serverless Puppet

- Begin with Resources and the RAL, where you'll learn about the fundamental building blocks of system configuration.
- After that, move on to Manifests and start controlling your system by writing actual Puppet code.
- Next, in Ordering, learn about dependencies and refresh events, manage the relationships between resources, and discover the most useful Puppet design pattern.
- In Variables, Conditionals, and Facts, make your manifests versatile by reading system information.
- In Classes and Modules, Part One, take the first step to a knowable and elegant site design and start turning your manifests into self-contained modules.

And come back soon, because there are more chapters on the way.

Chapter 3

Learning — Resources and the RAL

Resources are the building blocks of Puppet, and the division of resources into types and providers is what gives Puppet its power.

You are at the beginning. — Index — Manifests →

Molecules

Imagine a system's configuration as a collection of molecules; call them **"resources."**

These pieces vary in size, complexity, and lifespan: a user account can be a resource, as can a specific file, a software package, a running service, or a scheduled cron job. Even a single invocation of a shell command can be a resource.

Any resource is very similar to a class of related things: every file has a path and an owner, and every user has a name, a UID, and a group. Which is to say: *similar resources can be grouped into types.* Furthermore, the most important attributes of a resource type are usually conceptually identical across operating systems, regardless of how the implementations differ. That is, *the description of a resource can be abstracted away from its implementation.*

These two insights form Puppet's resource abstraction layer (RAL). The RAL splits resources into **types** (high-level models) and **providers** (platform-specific implementations), and lets you describe resources in a way that can apply to any system.

Sync: Read, Check, Write

Puppet uses the RAL to both read and modify the state of resources on a system. Since it's a declarative system, Puppet starts with an understanding of what state a resource *should* have. To sync the resource, it uses the RAL to query the current state, compares that against the desired state, then uses the RAL again to make any necessary changes.

Anatomy of a Resource

In Puppet, every resource is an instance of a **resource type** and is identified by a **title;** it has a number of **attributes** (which are defined by its type), and each attribute has a **value.**

The Puppet language represents a resource like this:

```
user { 'dave':
  ensure     => present,
  uid        => '507',
  gid        => 'admin',
  shell      => '/bin/zsh',
  home       => '/home/dave',
  managehome => true,
}
```

This syntax is the heart of the Puppet language, and you'll be seeing it a lot. Hopefully you can already see how it lays out all of the resource's parts (type, title, attributes, and values) in a fairly straightforward way.

The Resource Shell

Puppet ships with a tool called puppet resource, which uses the RAL to let you query and modify your system from the shell. Use it to get some experience with the RAL before learning to write and apply manifests.

Puppet resource's first argument is a resource type. If executed with no further arguments…

```
$ puppet resource user
```

… it will query the system and return every resource of that type it can recognize in the system's current state.

You can retrieve a specific resource's state by providing a resource name as a second argument.

```
$ puppet resource user root

user { 'root':
    home => '/var/root',
    shell => '/bin/sh',
    uid => '0',
    ensure => 'present',
    password => '*',
    gid => '0',
    comment => 'System Administrator'
}
```

Note that puppet resource returns Puppet code when it reads a resource from the system! You can use this code later to restore the resource to the state it's in now.

If any attribute=value pairs are provided as additional arguments to puppet resource, it will modify the resource, which can include creating it or destroying it:

```
$ puppet resource user dave ensure=present shell="/bin/zsh" home="/home/dave" managehome=true

notice: /User[dave]/ensure: created

user { 'dave':
    ensure => 'present',
    home => '/home/dave',
    shell => '/bin/zsh'
}
```

(Note that this command line assignment syntax differs from the Puppet language's normal attribute => value syntax.)

Finally, if you specify a resource and use the --edit flag, you can change that resource in your text editor; after the buffer is saved and closed, puppet resource will modify the resource to match your changes.

The Core Resource Types

Puppet has a number of built-in types, and new native types can be distributed with modules. Puppet's core types, the ones you'll get familiar with first, are notify, file, package, service, exec, cron, user, and group. Don't worry about memorizing them immediately, since we'll be covering various resources as we use them, but do take a second to print out a copy of the core types cheat sheet, a double-sided page covering these eight types. It is doctor-recommended[1] and has been clinically shown to treat reference inflammation.

Documentation for all of the built-in types can always be found in the reference section of this site, and can be generated on the fly with the puppet describe utility.

An Aside: puppet describe -s

You can get help for any of the Puppet executables by running them with the `--help` flag, but it's worth pausing for an aside on puppet describe's `-s` flag.

```
$ puppet describe -s user

user
====
Manage users.  This type is mostly built to manage system
users, so it is lacking some features useful for managing normal
users.

This resource type uses the prescribed native tools for creating
groups and generally uses POSIX APIs for retrieving information
about them.  It does not directly modify `/etc/passwd` or anything.


Parameters
----------

    allowdupe, auth_membership, auths, comment, ensure, expiry, gid, groups,
    home, key_membership, keys, managehome, membership, name, password,
    password_max_age, password_min_age, profile_membership, profiles,
    project, role_membership, roles, shell, uid

Providers
---------

    directoryservice, hpuxuseradd, ldap, pw, user_role_add, useradd
```

`-s` makes puppet describe dump a compact list of the given resource type's attributes and providers. This isn't useful when learning about a type for the first time or looking up allowed values, but it's fantastic when you have the name of an attribute on the tip of your tongue or you can't remember which two out of "group," "groups," and "gid" are applicable for the user type.

Next

Puppet resource can be useful for one-off jobs, but Puppet was born for greater things. Time to write some manifests.

1. The core types cheat sheet is not actually doctor-recommended. If you're a sysadmin with an M.D., please email me so I can change this footnote.

Chapter 4

Learning — Manifests

You understand the RAL; now learn about manifests and start writing and applying Puppet code.

← Resources and the RAL — Index — Resource Ordering →

No Strings Attached

You probably already know that Puppet usually runs in an agent/master (that is, client/server) configuration, but ignore that for now. It's not important yet and you can get a lot done without it, so for the time being, we have no strings on us.

Instead, we're going to use puppet apply, which applies a manifest on the local system. It's the simplest way to run Puppet, and it works like this:

```
$ puppet apply my_test_manifest.pp
```

Yeah, that easy.

You can use `puppet` — that is, without any subcommand — as a shortcut for `puppet apply`; it has the rockstar parking in the UI because of how often it runs at an interactive command line. I'll mostly be saying "puppet apply" for clarity's sake.

The behavior of Puppet's man pages is currently in flux. You can always get help for Puppet's command line tools by running the tool with the `--help` flag; in the Learning Puppet VM, which uses Puppet Enterprise, you can also run `pe-man puppet apply` to get the same help in a different format. Versions of Puppet starting with the upcoming 2.7 will use Git-style man pages (`man puppet-apply`) with improved formatting.

Manifests

Puppet programs are called "manifests," and they use the .pp file extension.

The core of the Puppet language is the *resource declaration,* which represents the desired state of one resource. Manifests can also use conditional statements, group resources into collections, generate text with functions, reference code in other manifests, and do many other things, but it all ultimately comes down to making sure the right resources are being managed the right way.

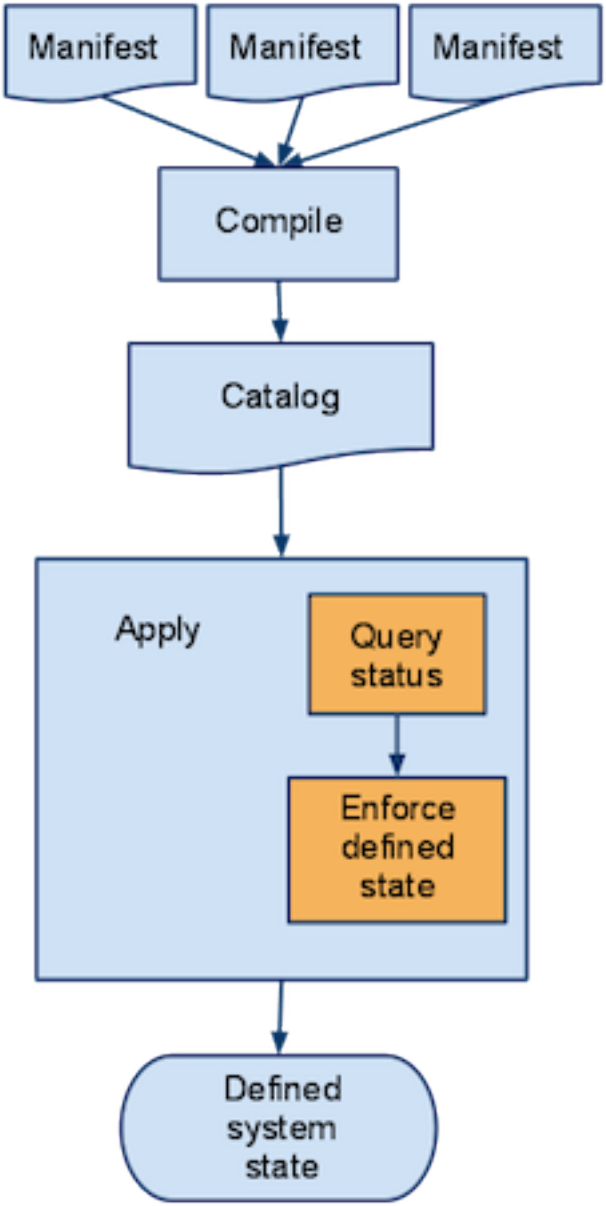

Figure 4.1: Diagram: Manifests are compiled into a catalog, which is then applied to yield the desired system state.

An Aside: Compilation

Manifests don't get used directly when Puppet syncs resources. Instead, the flow of a Puppet run goes a little like this:

Before being applied, manifests get compiled into a **"catalog,"** which is a directed acyclic graph that only represents resources and the order in which they need to be synced. All of the conditional logic, data lookup, variable interpolation, and resource grouping gets computed away during compilation, and the catalog doesn't have any of it.

Why? Several really good reasons, which we'll get to once we rediscover agent/master Puppet;[1] it's not urgent at the moment. But I'm mentioning it now as kind of an experiment: I think there are several things in Puppet that are easy to explain if you understand that split and quite baffling if you don't, so try keeping this in the back of your head and we'll see if it pays off later.

OK, enough about that; let's write some code! This will all be happening on your main Learning Puppet VM, so log in as root now; you'll probably want to stash these test manifests somewhere convenient, like `/root/learning-manifests`.

Resource Declarations

Let's start by just declaring a single resource:

```
# /root/training-manifests/1.file.pp

file {'testfile':
  path    => '/tmp/testfile',
  ensure  => present,
  mode    => 0640,
  content => "I'm a test file.",
}
```

And apply!

```
# puppet apply 1.file.pp
notice: /Stage[main]//File[testfile]/ensure: created
# cat /tmp/testfile
I'm a test file.
# ls -lah /tmp/testfile
-rw-r----- 1 root root 16 Feb 23 13:15 /tmp/testfile
```

You've seen this syntax before, but let's take a closer look at the language here.

- First, you have the **type** ("file"), followed by…
- …a pair of curly braces that encloses everything else about the resource. Inside those, you have…
  - …the resource **title,** followed by a colon…
  - …and then a set of **attribute => value** pairs describing the resource.

A few other notes about syntax:

- Missing commas and colons are the number one syntax error made by learners. If you take out the comma after `ensure => present` in the example above, you'll get an error like this:

  ```
  Could not parse for environment production: Syntax error at 'mode'; expected '}' at /root/manifests
      /1.file.pp:6 on node barn2.magpie.lan
  ```

  Missing colons do about the same thing. So watch for that. Also, although you don't strictly need the comma after the final attribute => value pair, you should always put it there anyhow. Trust me.

- Capitalization matters! You can't declare a resource with `File {'testfile :'...` , because that does something entirely different. (Specifically, it breaks. But it's *kind* of similar to what we use to tweak an existing resource, which we'll get to later.)

- Quoting values matters! Built-in values like present shouldn't be quoted, but normal strings should be. For all intents and purposes, everything is a string, including numbers. Puppet uses the same rules for single and double quotes as everyone else:
  - Single quotes are completely literal, except that you write a literal quote with \' and a literal backslash with \\.
  - Double quotes let you interpolate $variables and add newlines with \n.
- Whitespace is fungible for readability. Lining up the => arrows (sometimes called "fat commas") is good practice if you ever expect someone else to read this code — note that future and mirror universe versions of yourself count as "someone else."

Exercise: Declare a file resource in a manifest and apply it! Try changing the login message by setting the content of /etc/motd.

Once More, With Feeling!

Okay, you sort of have the idea by now. Let's make a whole wad of totally useless files! (And throw in some notify resources for good measure.)

```
# /root/training-manifests/2.file.pp

file {'/tmp/test1':
  ensure  => present,
  content => "Hi.",
}

file {'/tmp/test2':
  ensure => directory,
  mode   => 0644,
}

file {'/tmp/test3':
  ensure => link,
  target => '/tmp/test1',
}

notify {"I'm notifying you.":} # Whitespace is fungible, remember.
notify {"So am I!":}
```

```
# puppet apply 2.file.pp
notice: /Stage[main]//File[/tmp/test2]/ensure: created
notice: /Stage[main]//File[/tmp/test3]/ensure: created
notice: /Stage[main]//File[/tmp/test1]/ensure: created
notice: I'm notifying you.
notice: /Stage[main]//Notify[I'm notifying you.]/message: defined 'message' as 'I'm notifying you.'
notice: So am I!
notice: /Stage[main]//Notify[So am I!]/message: defined 'message' as 'So am I!'

# ls -lah /tmp/test*
-rw-r--r--  1 root root    3 Feb 23 15:54 test1
lrwxrwxrwx  1 root root   10 Feb 23 15:54 test3 -> /tmp/test1
-rw-r-----  1 root root   16 Feb 23 15:05 testfile

/tmp/test2:
total 16K
drwxr-xr-x 2 root root 4.0K Feb 23 16:02 .
drwxrwxrwt 5 root root 4.0K Feb 23 16:02 ..

# cat /tmp/test3
Hi.
```

That was totally awesome. What just happened?

Titles and Namevars

All right, notice how we left out some important attributes there and everything still worked? Almost every resource type has one attribute whose value defaults to the resource's title. For the file resource, that's path; with notify, it's message. A lot of the time (user, group, package…), it's plain old name.

To people who occasionally delve into the Puppet source code, the one attribute that defaults to the title is called the **"namevar,"** which is a little weird but as good a name as any. It's almost always the attribute that amounts to the resource's *identity,* the one thing that should always be unique about each instance.

This can be a convenient shortcut, but be wary of overusing it; there are several common cases where it makes more sense to give a resource a symbolic title and assign its name (-var) as a normal attribute. In particular, it's a good idea to do so if a resource's name is long or you want to assign the name conditionally depending on the nature of the system.

```
notify {'bignotify':
  message => "I'm completely enormous, and will mess up the formatting of your
      code! Also, since I need to fire before some other resource, you'll need
      to refer to me by title later using the Notify['title'] syntax, and you
      really don't want to have to type this all over again.",
}
```

The upshot is that our notify {"I'm notifying you.":} resource above has the exact same effect as:

```
notify {'other title':
  message => "I'm notifying you.",
}
```

… because the message attribute just steals the resource title if you don't give it anything of its own.

You can't declare the same resource twice: Puppet will always keep you from making resources with duplicate titles, and will almost always keep you from making resources with duplicate name/namevar values. (exec resources are the main exception.)

And finally, you don't need an encyclopedic memory of what the namevar is for each resource — when in doubt, just choose a descriptive title and specify the attributes you need.

644 = 755 For Directories

We said /tmp/test2/ should have permissions mode 0644, but our ls -lah showed mode 0755. That's because Puppet groups the read bit and the traverse bit for directories, which is almost always what you actually want. The idea is to let you recursively manage whole directories as mode 0644 without making all their files executable.

New Ensure Values

The file type has several different values for its ensure attribute: present, absent, file, directory, and link. They're listed on the core types cheat sheet whenever you need to refresh your memory, and they're fairly self-explanatory.

The Destination

Here's a pretty crucial part of learning to think like a Puppet user. Try applying that manifest again.

```
# puppet apply 2.file.pp
notice: I'm notifying you.
notice: /Stage[main]//Notify[I'm notifying you.]/message: defined 'message' as 'I'm notifying you.'
notice: So am I!
notice: /Stage[main]//Notify[So am I!]/message: defined 'message' as 'So am I!'
```

And again!

```
# rm /tmp/test3
# puppet apply 2.file.pp
notice: I'm notifying you.
notice: /Stage[main]//Notify[I'm notifying you.]/message: defined 'message' as 'I'm notifying you.'
```

```
notice: /Stage[main]//File[/tmp/test3]/ensure: created
notice: So am I!
notice: /Stage[main]//Notify[So am I!]/message: defined 'message' as 'So am I!'
```

The notifies are firing every time, because that's what they're for, but Puppet doesn't do anything with the file resources unless they're wrong on disk; if they're wrong, it makes them right. Remember how I said Puppet was declarative? This is how that pays off: You can apply the same configuration every half hour without having to know anything about how the system currently looks. Manifests describe the destination, and Puppet handles the journey.

> **Exercise:** Write and apply a manifest that'll make sure Apache (`httpd`) is running, use a web browser on your host OS to view the Apache welcome page, then modify the manifest to turn Apache back off. (Hint: You'll have to check the cheat sheet or the types reference, because the `service` type's `ensure` values differ from the ones you've seen so far.)

> **Slightly more difficult exercise:** Write and apply a manifest that uses the `ssh_authorized_key` type to let you log into the learning VM as root without a password. You'll need to already have an SSH key.

Next

Resource declarations: Check! You know how to use the fundamental building blocks of Puppet code, so now it's time to learn how those blocks fit together.

1. There are also a few I can mention now, actually. If you drastically refactor your manifest code and want to make sure it still generates the same configurations, you can just intercept the catalogs and use a special diff tool on them; if the same nodes get the same configurations, you can be sure the code acts the same without having to model the execution of the code in your head. Compiling to a catalog also makes it much easier to simulate applying a configuration, and since the catalog is just data, it's relatively easy to parse and analyze with your own tool of choice.

Chapter 5

Learning — Resource Ordering

You understand manifests and resource declarations; now learn about metaparameters, resource ordering, and one of the most useful patterns in Puppet.

← Manifests — Index — Variables →

Disorder

Let's look back on one of our manifests from the last page:

```
# /root/training-manifests/2.file.pp

file {'/tmp/test1':
  ensure  => present,
  content => "Hi.",
}

file {'/tmp/test2':
  ensure => directory,
  mode   => 644,
}

file {'/tmp/test3':
  ensure => link,
  target => '/tmp/test1',
}

notify {"I'm notifying you.":}
notify {"So am I!":}
```

Although we wrote these declarations one after another, Puppet might sync them in any order: unlike with a procedural language, the physical order of resources in a manifest doesn't imply a logical order.

But some resources depend on other resources. So how do we tell Puppet which ones go first?

Metaparameters, Resource References, and Ordering

```
file {'/tmp/test1':
  ensure  => present,
  content => "Hi.",
}

notify {'/tmp/test1 has already been synced.':
```

```
  require => File['/tmp/test1'],
}
```

Each resource type has its own set of attributes, but there's another set of attributes, called metaparameters, which can be used on any resource. (They're meta because they don't describe any feature of the resource that you could observe on the system after Puppet finishes; they only describe how Puppet should act.)

There are four metaparameters that let you arrange resources in order: before, require, notify, and subscribe. All of them accept a **resource reference** (or an array[1] of them). Resource references look like this:

```
Type['title']
```

(Note the square brackets and capitalized resource type!)

An Aside: Capitalization

The easy way to remember this is that you *only* use the lowercase type name when declaring a new resource. Any other situation will always call for a capitalized type name.

This will get more important in another couple lessons, so I'll mention it again later.

Before and Require

before and require make simple dependency relationships, where one resource must be synced before another. before is used in the earlier resource, and lists resources that depend on it; require is used in the later resource and lists the resources that it depends on.

These two metaparameters are just different ways of writing the same relationship — our example above could just as easily be written like this:

```
file {'/tmp/test1':
  ensure  => present,
  content => "Hi.",
  before  => Notify['/tmp/test1 has already been synced.'],
  # (See what I meant about symbolic titles being a good idea?)
}

notify {'/tmp/test1 has already been synced.':}
```

Notify and Subscribe

A few resource types[2] can be **"refreshed"** — that is, told to react to changes in their environment. For a service, this usually means restarting when a config file has been changed; for an exec resource, this could mean running its payload if any user accounts have been changed. (Note that refreshes are performed by Puppet, so they only occur during Puppet runs.)

The notify and subscribe metaparameters make dependency relationships the way before and require do, but they also make **refresh relationships.** Not only will the earlier resource in the pair get synced first, but if Puppet makes any changes to that resource, it will send a refresh event to the later resource, which will react accordingly.

Chaining

```
file {'/tmp/test1':
  ensure  => present,
  content => "Hi.",
}

notify {'after':
  message => '/tmp/test1 has already been synced.',
}

File['/tmp/test1'] -> Notify['after']
```

There's one last way to declare relationships: chain resource references with the ordering (->) and notification (~>; note the tilde) arrows. The arrows can point in either direction (<- works too), and you should think

of them as representing the flow of time: the resource at the blunt end of the arrow will be synced *before* the resource the arrow points at.

The example above yields the same dependency as the two examples before it. The benefit of this alternate syntax may not be obvious when we're working with simple examples, but it can be much more expressive and legible when we're working with resource collections.

Autorequire

Some of Puppet's resource types will notice when an instance is related to other resources, and they'll set up automatic dependencies. The one you'll use most often is between files and their parent directories: if a given file and its parent directory are both being managed as resources, Puppet will make sure to sync the parent directory before the file.

Don't sweat much about the details of autorequiring; it's fairly conservative and should generally do the right thing without getting in your way. If you forget it's there and make explicit dependencies, your code will still work.

Summary

So to sum up: whenever a resource depends on another resource, use the `before` or `require` metaparameter or chain the resources with `->`. Whenever a resource needs to refresh when another resource changes, use the `notify` or `subscribe` metaparameter or chain the resources with `~>`. Some resources will autorequire other resources if they see them, which can save you some effort.

Hopefully that's all pretty clear! But even if it is, it's rather abstract — making sure a notify fires after a file is something of a "hello world" use case, and not very illustrative. Let's break something!

Example: sshd

You've probably been using SSH and your favorite terminal app to interact with the Learning Puppet VM, so let's go straight for the most-annoying-case scenario: we'll pretend someone accidentally gave the wrong person (i.e., us) sudo privileges, and have you ruin root's ability to SSH to this box. We'll use Puppet to bust it and Puppet to fix it.

First, if you got the `ssh_authorized_key` exercise from the last page working, undo it.

```
# mv ~/.ssh/authorized_keys ~/old_ssh_authorized_keys
```

Now let's get a copy of the current sshd config file; going forward, we'll use our new copy as the canonical source for that file.

```
# cp /etc/ssh/sshd_config ~/learning-manifests/
```

Next, edit our new copy of the file. There's a line in there that says `PasswordAuthentication yes`; find it, and change the yes to a no. Then start writing some Puppet!

```
# /root/learning-manifests/break_ssh.pp
file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => 600,
  source => '/root/learning-manifests/sshd_config',
  # And yes, that's the first time we've seen the "source" attribute.
  # It accepts absolute paths and puppet:/// URLs, about which more later.
}
```

Except that won't work! (Don't run it, and if you did, read this footnote.[3]) If we apply this manifest, the config file will change, but `sshd` will keep acting on the old config file until it restarts… and if it's only restarting when the system reboots, that could be years from now.

If we want the service to change its behavior as soon as we change our policy, we'll have to tell it to monitor the config file.

```
# /root/learning-manifests/break_ssh.pp, again
file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => 600,
  source => '/root/learning-manifests/sshd_config',
}

service { 'sshd':
  ensure     => running,
  enable     => true,
  hasrestart => true,
  hasstatus  => true,
  # FYI, those last two attributes default to false, since
  # bad init scripts are more or less endemic.
  subscribe  => File['/etc/ssh/sshd_config'],
}
```

And that'll do it! Run that manifest with puppet apply, and after you log out, you won't be able to SSH into the VM again. *Victory.*

To fix it, you'll have to log into the machine directly — use the screen provided by your virtualization app. Once you're there, you'll just have to edit /root/learning-manifests/sshd_config again to change the PasswordAuthentication setting and re-apply the same manifest; Puppet will replace /etc/ssh/sshd_config with the new version, restart the service, and re-enable remote password logins. (And you can put your SSH key back now, if you like.)

Package/File/Service

The example we just saw was very close to a pattern you'll see constantly in production Puppet code, but it was missing a piece. Let's complete it:

```
# /root/learning-manifests/break_ssh.pp
package { 'openssh-server':
  ensure => present,
  before => File['/etc/ssh/sshd_config'],
}

file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => 600,
  source => '/root/learning-manifests/sshd_config',
}

service { 'sshd':
  ensure     => running,
  enable     => true,
  hasrestart => true,
  hasstatus  => true,
  subscribe  => File['/etc/ssh/sshd_config'],
}
```

This is **package/file/service,** one of the most useful patterns in Puppet: the package resource makes sure the software is installed, the config file depends on the package resource, and the service subscribes to changes in the config file.

It's hard to understate the importance of this pattern; if this was all you knew how to do with Puppet, you could still do a fair amount of work. But we're not done yet.

Next

Now that you can sync resources in their proper order, it's time to make your manifests aware of the outside world with variables, facts, and conditionals.

1. Arrays in Puppet are made with square brackets and commas, so an array of resource references would [Notify['look'], Notify['like'], Notify['this']].
2. Of the built-in types, only exec, service, and mount can be refreshed.

3. If you DID apply the incomplete manifest, something interesting happened: your machine is now in a half-rolled-out condition that puts the lie to what I said earlier about not having to worry about the system's current state. Since the config file is now in sync with its desired state, Puppet won't change it during the next run, which means applying the complete manifest won't cause the service to refresh until either the source file or the file on the system changes one more time.

   In practice, this isn't a huge problem, because only your development machines are likely to end up in this state; your production nodes won't have been given incomplete configurations. In the meantime, you have two options for cleaning up after applying an incomplete manifest: For a one-time fix, echo a bogus comment to the bottom of the file on the system (`echo "# ignoreme" >> /etc/ssh/sshd_config`), or for a more complete approach, make a comment in the source file that contains a version string, which you can update whenever you make significant changes to the associated manifest(s). Both of these approaches will mark the config file as out of sync, replace it during the Puppet run, and send the refresh event to the service.

Chapter 6

Learning — Variables, Conditionals, and Facts

You can write manifests and order resources; now, add logic and flexibility with conditional statements and variables.

← Ordering — Index — Modules (Part One) →

Variables

Variables! I'm going to bet you pretty much know this drill, so let's move a little faster:

- `$variables` always start with a dollar sign. You assign to variables with the = operator.
- Variables can hold strings, numbers, special values (false, undef…), arrays, and hashes.
- You can use variables as the value for any resource attribute, or as the title of a resource.
- You can also interpolate variables inside strings, if you use double-quotes. To distinguish a `${variable}` from the surrounding text, you should wrap its name in curly braces.
- Every variable has a short local name and a long fully-qualified name. Fully qualified variables look like `$scope::variable`. Top scope variables are the same, but their scope is nameless. (For example: `$::top_scope_variable`.)
- You can only assign the same variable **once** in a given scope.[1]

```
$longthing = "Imagine I have something really long in here. Like an SSH key, let's say."

file {'authorized_keys':
  path    => '/root/.ssh/authorized_keys',
  content => $longthing,
}
```

Pretty easy.

Facts

And now, a teaspoon of magic.

Before you even start writing your manifests, Puppet builds you a stash of pre-assigned variables. Check it out:

```
# hosts-simple.pp

# Host type reference:
# http://docs.puppetlabs.com/references/stable/type.html#host

host {'self':
  ensure       => present,
  name         => $::fqdn,
  host_aliases => ['puppet', $::hostname],
  ip           => $::ipaddress,
}

file {'motd':
  ensure  => file,
  path    => '/etc/motd',
  mode    => 0644,
  content => "Welcome to ${::hostname},\na ${::operatingsystem} island in the sea of ${::domain}.\n",
}
```

```
# puppet apply hosts-simple.pp

notice: /Stage[main]//Host[puppet]/ensure: created
notice: /Stage[main]//File[motd]/ensure: defined content as '{md5}d149026e4b6d747ddd3a8157a8c37679'

# cat /etc/hosts
# HEADER: This file was autogenerated at Mon Apr 25 14:39:11 -0700 2011
# HEADER: by puppet.  While it can still be managed manually, it
# HEADER: is definitely not recommended.
# Do not remove the following line, or various programs
# that require network functionality will fail.
127.0.0.1   localhost.localdomain   localhost
::1 localhost6.localdomain6 localhost6
172.16.158.137  learn.puppet.demo     puppet  learn
```

Our manifests are starting to get versatile, with pretty much no real work on our part.

Hostname? IPaddress?

So where did those helpful variables come from? They're "facts." Puppet ships with a tool called Facter, which ferrets out your system information, normalizes it into a set of variables, and passes them off to Puppet. The compiler then has access to those facts when it's reading a manifest.

There are a lot of different facts, and the easiest way to get a list of them is to simply run `facter` at your VM's command line. You'll get back a long list of key/value pairs separated by the familiar => hash rocket. To use one of these facts in your manifests, just prepend a dollar sign to its name (along with a ::, because being explicit about namespaces is a good habit).

Most kinds of system will have at least a few facts that aren't available on other kinds of system (e.g., try comparing Facter's output on your CentOS VM to what it does on an OS X machine), and it can get fuzzier if you're extending Facter with custom facts, but there's a general core of facts that give you the same info everywhere. You'll get a feel for them pretty quickly, and can probably guess most of them just by reading the list of names.

Conditional Statements

Puppet has a fairly complete complement of conditional syntaxes, and the info available in facts makes it really easy to code different behavior for different systems.

If

We'll start with your basic if statement. Same as it ever was: ***if*** *condition { block of code }* ***elsif*** *condition { block of code }* ***else*** *{ block of code };* the else and any number of elsif statements are optional.

```
if $is_virtual {
  service {'ntpd':
    ensure => stopped,
    enable => false,
  }
}
else {
  service { 'ntpd':
    name       => 'ntpd',
    ensure     => running,
    enable     => true,
    hasrestart => true,
    require => Package['ntp'],
  }
}
```

The blocks of code for each condition can contain any Puppet code.

What is False?

You'll notice I used a bare fact as the condition above. The Puppet language's data types are kind of loose, and a lot of things tend to get represented internally as strings, so it's worth mentioning that the following values will be treated as false by an if statement:

- undef
- '' (the empty string)
- false
- Any expression that evaluates to false.

In particular, be aware that 0 is true.

Conditions

Conditions can get pretty sophisticated: you can use any valid expression in an if statement. Usually, this is going to mean using one of the standard comparison operators (==, !=, <, >, <=, >=), the regex match operators (=~ and !~), or the in operator (which tests whether the right operand contains the left one).

Case

Also probably familiar: the case statement. (Or switch, or whatever your language of choice calls it.)

```
case $operatingsystem {
  centos: { $apache = "httpd" }
  # Note that these matches are case-insensitive.
  redhat: { $apache = "httpd" }
  debian: { $apache = "apache2" }
  ubuntu: { $apache = "apache2" }
  default: { fail("Unrecognized operating system for webserver") }
  # "fail" is a function. We'll get to those later.
}
package {'apache':
  name   => $apache,
  ensure => latest,
}
```

Instead of testing a condition up front, case matches a variable against a bunch of possible values. **`default` is a special value,** which does exactly what it sounds like.

Case matching

Matches can be simple strings (like above), regular expressions, or comma-separated lists of either.

String matching is case-insensitive, like the == comparison operator. Regular expressions are denoted with the slash-quoting used by Perl and Ruby; they're case-sensitive by default, but you can use the (?i) and (?-i) switches to turn case-insensitivity on and off inside the pattern. Regex matches also assign captured subpatterns to $1, $2, etc. inside the associated code block, with $0 containing the whole matching string.

Here's a regex example:

```
case $ipaddress_eth0 {
  /^127[\d.]+$/: {
    notify {'misconfig':
      message => "Possible network misconfiguration: IP address of $0",
    }
  }
}
```

And here's the example from above, rewritten and more readable:

```
case $operatingsystem {
  centos, redhat: { $apache = "httpd" }
  debian, ubuntu: { $apache = "apache2" }
  default: { fail("Unrecognized operating system for webserver") }
}
```

Selectors

Selectors might be less familiar; they're kind of like the ternary operator, and kind of like the case statement.

Instead of choosing between a set of code blocks, selectors choose between a group of possible values. You can't use them on their own; instead, they're usually used to assign a variable.

```
$apache = $operatingsystem ? {
  centos                => 'httpd',
  redhat                => 'httpd',
  /(?i)(ubuntu|debian)/ => "apache2-$1",
    # (Don't actually use that package name.)
  default               => undef,
}
```

Careful of the syntax, there: it looks kind of like we're saying $apache = $operatingsystem, but we're not. The question mark flags $operatingsystem as the pivot of a selector, and the actual value that gets assigned is determined by which option $operatingsystem matches. Also note how the syntax differs from the case syntax: it uses hash rockets and line-end commas instead of colons and blocks, and you can't use lists of values in a match. (If you want to match against a list, you have to fake it with a regular expression.)

It can look a little awkward, but there are plenty of situations where it's the most concise way to get a value sorted out; if you're ever not comfortable with it, you can just use a case statement to assign the variable instead.

Selectors can also be used directly as values for a resource attribute, but try not to do that, because it gets ugly fast.

Exercises

Exercise: Use the $operatingsystem fact to write a manifest that installs a build environment on Debian-based ("debian" and "ubuntu") and Enterprise Linux-based ("centos," "redhat") machines. (Both types of system require the gcc package, but Debian-type systems also require build-essential.)

Exercise: Write a manifest that installs and configures NTP for Debian-based and Enterprise Linux-based Linux systems. This will be a package/file/service pattern where you'll be shipping different

config files (Debian version, Red Hat version — remember the file type's "source" attribute) and using different service names (ntp and ntpd, respectively).

(Use a second manifest to disable the NTP service after you've gotten this example working; NTP can behave kind of uselessly in a virtual machine.)

Next

Now that your manifests can adapt to different kinds of systems, it's time to start grouping resources and conditionals into meaningful units. Onward to classes, defined resource types, and modules!

1. This has to do with the declarative nature of the Puppet language: the idea is that the order in which you read the file shouldn't matter, so changing a value halfway through is illegal, since it would make the results order-dependent.

   In practice, this isn't the full story, because you can't currently read a variable from anywhere north of its assignment. We're working on that.

Chapter 7

Learning — Modules and Classes (Part One)

You can write some pretty sophisticated manifests at this point, but they're still at a fairly low altitude, going resource-by-resource-by-resource. Now, zoom out with resource collections.

← Variables, etc. — Index — TBA →

Collecting and Reusing

At some point, you're going to have Puppet code that fits into a couple of different buckets: really general stuff that applies to all your machines, more specialized stuff that only applies to certain classes of machines, and very specific stuff that's meant for a few nodes at most.

So… you *could* just paste in all your more general code as boilerplate atop your more specific code. There are ways to do that and get away with it. But that's the road down into the valley of the 4,000-line manifest. Better to separate your code out into meaningful units and then call those units by name as needed.

Thus, resource collections and modules! In a few minutes, you'll be able to maintain your manifest code in one place and declare whole groups of it like this:

```
class {'security_base': }
class {'webserver_base': }
class {'appserver': }
```

And after that, it'll get even better. But first things first.

Classes

Classes are singleton collections of resources that Puppet can apply as a unit. You can think of them as blocks of code that can be turned on or off.

If you know any object-oriented programming, try to ignore it for a little while, because that's not the kind of class we're talking about. Puppet classes could also be called "roles" or "aspects;" they describe one part of what makes up a system's identity.

Defining

Before you can use a class, you have to **define** it, which is done with the `class` keyword, a name, and a block of code:

```
class someclass {
  ...
}
```

Well, hey: you have a block of code hanging around from last chapter's exercises, right? May as well just wrap *that* in a class definition!

```
# ntp-class1.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file    = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file    = 'ntp.conf.debian'
    }
  }

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

  service { 'ntp':
    name      => $service_name,
    ensure    => running,
    enable    => true,
    subscribe => File['ntp.conf'],
  }

  file { 'ntp.conf':
    path    => '/etc/ntp.conf',
    ensure  => file,
    require => Package['ntp'],
    source  => "/root/learning-manifests/${conf_file}",
  }
}
```

Go ahead and apply that. In the meantime:

An Aside: Names, Namespaces, and Scope

Class names have to start with a lowercase letter, and can contain lowercase alphanumeric characters and underscores. (Just your standard slightly conservative set of allowed characters.)

Class names can also use a double colon (::) as a namespace separator. (Yes, this should look familiar.) This is a good way to show which classes are related to each other; for example, you can tell right away that something's going on between apache::ssl and apache::vhost. This will become more important about two feet south of here.

Also, class definitions introduce new variable scopes. That means any variables you assign within won't be accessible by their short names outside the class; to get at them from elsewhere, you would have to use the fully-qualified name (e.g. $apache::ssl::certificate_expiration). It also means you can localize — mask — variable short names in use outside the class; if you assign a $fqdn variable in a class, you would get the new value instead of the value of the Facter-supplied variable, unless you used the fully-qualified fact name ($::fqdn).

Declaring

Okay, back to our example, which you'll have noticed by now doesn't actually *do* anything.

```
# puppet apply ntp-class1.pp
(...silence)
```

The code inside the class was properly parsed, but the compiler didn't build any of it into the catalog, so none of the resources got synced. For that to happen, the class has to be declared.

You actually already know the syntax to do that. A class definition just enables a unique instance of the class resource type, so you can declare it like any other resource:

```
# ntp-class1.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file    = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file    = 'ntp.conf.debian'
    }
  }

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

  service { 'ntp':
    name      => $service_name,
    ensure    => running,
    enable    => true,
    subscribe => File['ntp.conf'],
  }

  file { 'ntp.conf':
    path    => '/etc/ntp.conf',
    ensure  => file,
    require => Package['ntp'],
    source  => "/root/learning-manifests/${conf_file}",
  }
}

# Then, declare it:
class {'ntp': }
```

This time, all those resources will end up in the catalog:

```
# puppet apply --verbose ntp-class1.pp

info: Applying configuration version '1305066883'
info: FileBucket adding /etc/ntp.conf as {md5}5baec8bdbf90f877a05f88ba99e63685
info: /Stage[main]/Ntp/File[ntp.conf]: Filebucketed /etc/ntp.conf to puppet with sum 5
     baec8bdbf90f877a05f88ba99e63685
notice: /Stage[main]/Ntp/File[ntp.conf]/content: content changed '{md5}5baec8bdbf90f877a05f88ba99e63685'
     to '{md5}dc20e83b436a358997041a4d8282c1b8'
info: /Stage[main]/Ntp/File[ntp.conf]: Scheduling refresh of Service[ntp]
notice: /Stage[main]/Ntp/Service[ntp]/ensure: ensure changed 'stopped' to 'running'
notice: /Stage[main]/Ntp/Service[ntp]: Triggered 'refresh' from 1 events
```

Defining the class makes it available; declaring activates it.

Include

There's another way to declare classes, but it behaves a little bit differently:

```
include ntp
include ntp
include ntp
```

The include function will declare a class if it hasn't already been declared, and will do nothing if it has. This means you can safely use it multiple times, whereas the resource syntax can only be used once. The drawback is that include can't currently be used with parameterized classes, on which more later.

So which should you choose? Neither, yet: learn to use both, and decide later, after we've covered site design and parameterized classes.

Classes In Situ

You've probably already guessed that classes aren't enough: even with the code above, you'd still have to paste the ntp definition into all your other manifests. So it's time to meet the **module autoloader!**

An Aside: Printing Config

But first, we'll need to meet its friend, the modulepath.

```
# puppet apply --configprint modulepath
/etc/puppetlabs/puppet/modules
```

By the way, --configprint is basically my favorite. Puppet has a *lot* of config options, all of which have default values and site-specific overrides in puppet.conf, and trying to memorize them all is a pain. You can use --configprint on most of the Puppet tools, and they'll print a value (or a bunch, if you use --configprint all) and exit.

Modules

So anyway, modules are re-usable bundles of code and data. Puppet autoloads manifests from the modules in its modulepath, which means you can declare a class stored in a module from anywhere. Let's just convert that last class to a module immediately, so you can see what I'm talking about:

```
# cd /etc/puppetlabs/puppet/modules
# mkdir ntp; cd ntp; mkdir manifests; cd manifests
# vim init.pp
```

```
# init.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file    = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file    = 'ntp.conf.debian'
    }
  }

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

  service { 'ntp':
    name      => $service_name,
    ensure    => running,
    enable    => true,
    subscribe => File['ntp.conf'],
  }

  file { 'ntp.conf':
    path    => '/etc/ntp.conf',
    ensure  => file,
    require => Package['ntp'],
    source  => "/root/learning-manifests/${conf_file}",
  }
}

# (Remember not to declare the class yet.)
```

And now, the reveal:[1]

```
# cd ~
# puppet apply -e "include ntp"
```

It just works. You can now do that from any manifest, without having to cut and paste anything.

But we're not quite done yet. See how the manifest is referring to some files stored outside the module? Let's fix that:

```
# mkdir /etc/puppetlabs/modules/ntp/files
# mv /root/learning-manifests/ntp.conf.* /etc/puppetlabs/modules/ntp/files/
# vim /etc/puppetlabs/modules/ntp/manifests/init.pp
```

```
# ...
  file { 'ntp.conf':
    path    => '/etc/ntp.conf',
    ensure  => file,
    require => Package['ntp'],
    # source => "/root/learning-manifests/${conf_file}",
    source  => "puppet:///modules/ntp/${conf_file}",
  }
}
```

There — our little example from last chapter has grown up into a self-contained blob of awesome.

Module Structure

A module is just a directory with stuff in it, and the magic comes from putting that stuff where Puppet expects to find it. Which is to say, arranging the contents like this:

- {module}/
  - files/
  - lib/
  - manifests/
    - init.pp
    - {class}.pp
    - {defined type}.pp
    - {namespace}/
      - {class}.pp
      - {class}.pp
  - templates/
  - tests/

The main directory should be named after the module. All of the manifests go in the `manifests` directory. Each manifest contains only one class (or defined type). There's a special manifest called `init.pp` that holds the module's main class, which should have the same name as the module. That's your barest-bones module: main folder, manifests folder, init.pp, just like we used in the ntp module above.

But if that was all a module was, it'd make more sense to just load your classes from one flat folder. Modules really come into their own with namespacing and grouping of classes.

Manifests, Namespacing, and Autoloading

The manifests directory can hold any number of other classes and even folders of classes, and Puppet uses namespacing to find them. Say we have a manifests folder that looks like this:

- foo/
  - manifests/
    - init.pp
    - bar.pp
    - bar/
      - baz.pp

The init.pp file should contain `class foo { ... }`, bar.pp should contain `class foo::bar { ... }`, and baz.pp should contain `class foo::bar::baz { ... }`.

This can be a little disorienting at first, but I promise you'll get used to it. Basically, init.pp is special, and all of the other classes (each in its own manifest) should be under the main class's namespace. If you add more levels of directory hierarchy, they get interpreted as more levels of namespace hierarchy.

Files

Puppet can serve files from modules, and it works identically regardless of whether you're doing serverless or agent/master Puppet. Everything in the `files` directory in the ntp module is available under the `puppet:///modules/ntp/` URL. Likewise, a `test.txt` file in the `testing` module's `files` could be retrieved as `puppet:///modules/testing/test.txt`.

Tests

Once you start writing modules you plan to keep for more than a day or two, read our brief guide to module smoke testing. It's pretty simple, and will eventually pay off.

Templates

More on templates later.

Lib

Puppet modules can also serve executable Ruby code from their `lib` directories, to extend Puppet and Facter. (Remember how I mentioned extending Facter with custom facts? This is where they live.) It'll be a while before we cover any of that.

Module Scaffolding

Since you'll be dealing with those same five subdirectories so much, I suggest adding a function for them to your .bashrc file.

```
mkmod() {
    mkdir "$1"
    mkdir "$1/files" "$1/lib" "$1/manifests" "$1/templates" "$1/tests"
}
```

Exercises

> **Exercise:** Build an Apache2 module and class, which ensures Apache is installed and running and manages its config file. While you're at it, make Puppet manage the DocumentRoot and put a custom 404 page and index.html in place.
>
> Set any files or package/service names that might vary per distro conditionally, failing if we're not on CentOS; this'll let you cleanly shim in support for other distros once you need it.
>
> We'll be using this module some more in future lessons. Oh yes.

Next

And we've reached another brief pause! There's some fun stuff ahead: come back next update, where we'll cover defined resource types, classes with parameters in 'em, and possibly inheritance, templates, functions, and/or resource defaults.

1. The `-e` flag lets you give puppet apply a line of manifest code instead of a file, same as with Perl or Ruby.

Chapter 8

Tools

This guide covers the major tools that comprise Puppet.

Single binary

From version 2.6.0 and onwards all the Puppet functions are also available via a single Puppet binary, in the style of git.

List of binary changes:

- puppetmasterd → puppet master
- puppetd → puppet agent
- puppet → puppet apply
- puppetca → puppet cert
- ralsh → puppet resource
- puppetrun → puppet kick
- puppetqd → puppet queue
- filebucket → puppet filebucket
- puppetdoc → puppet doc
- pi → puppet describe

This also results in a change in the puppet.conf configuration file. The sections, previously things like [puppetd], now should be renamed to match the new binary names. So [puppetd] becomes [agent]. You will be prompted to do this when you start Puppet. A log message will be generated for each section that needs to be renamed. This is merely a warning – existing configuration file will work unchanged.

Manpage documentation

Additional information about each tool is provided in the relevant manpage. You can consult the local version of each manpage, or view the web versions of the manuals.

puppet master (or puppetmasterd)

Puppet master is a central management daemon. In most installations, you'll have one puppet master server and each managed machine will run puppet agent. By default, puppet master operates a certificate authority, which can be managed using puppet cert.

Puppet master serves compiled configurations, files, templates, and custom plugins to managed nodes.

The main configuration file for puppet master, puppet agent, and puppet apply is /etc/puppet/puppet.conf, which has sections for each application.

puppet agent (or puppetd)

Puppet agent runs on each managed node. By default, it will wake up every 30 minutes (configurable), check in with puppetmasterd, send puppetmasterd new information about the system (facts), and receive a 'compiled catalog' describing the desired system configuration. Puppet agent is then responsible for making the system match the compiled catalog. If pluginsync is enabled in a given node's configuration, custom plugins stored on the Puppet Master server are transferred to it automatically.

The puppet master server determines what information a given managed node should see based on its unique identifier ("certname"); that node will not be able to see configurations intended for other machines.

puppet apply (or puppet)

When running Puppet locally (for instance, to test manifests, or in a non-networked disconnected case), puppet apply is run instead of puppet agent. It then uses local files, and does not try to contact the central server. Otherwise, it behaves the same as puppet agent.

puppet cert (or puppetca)

The puppet cert command is used to sign, list and examine certificates used by Puppet to secure the connection between the Puppet master and agents. The most common usage is to sign the certificates of Puppet agents awaiting authorisation:

```
> puppet cert --list
agent.example.com

> puppet cert --sign agent.example.com
```

You can also list all signed and unsigned certificates:

```
> puppet cert --all and --list
+ agent.example.com
agent2.example.com
```

Certificates with a + next to them are signed. All others are awaiting signature.

puppet doc (or puppetdoc)

Puppet doc generates documentation about Puppet and your manifests, which it can output in HTML, Markdown and RDoc.

puppet resource (or ralsh)

Puppet resource (also known as ralsh, for "Resource Abstraction Layer SHell") uses Puppet's resource abstraction layer to interactively view and manipulate your local system.

For example, to list information about the user 'xyz':

```
> puppet resource User "xyz"

user { 'xyz':
   home => '/home/xyz',
   shell => '/bin/bash',
   uid => '1000',
   comment => 'xyz,,,',
   gid => '1000',
   groups => ['adm','dialout','cdrom','sudo','plugdev','lpadmin','admin','sambashare','libvirtd'],
   ensure => 'present'
}
```

It can also be used to make additions and removals, as well as to list resources found on a system:

```
> puppet resource User "bob" ensure=present group=admin

notice: /User[bob]/ensure: created
user { 'bob':
    shell => '/bin/sh',
    home => '/home/bob',
    uid => '1001',
    gid => '1001',
    ensure => 'present',
    password => '!'
}

> puppet resource User "bob" ensure=absent
...

> puppet resource  User
...
```

Puppet resource is most frequently used as a learning tool, but it can also be used to avoid memorizing differences in common commands when maintaining multiple platforms. (Note that puppet resource can be used the same way on OS X as on Linux, e.g.)

puppet inspect

Puppet inspect generates an inspection report and sends it to the puppet master. It cannot be run as a daemon.

Inspection reports differ from standard Puppet reports, as they do not record the actions taken by Puppet when applying a catalog; instead, they document the current state of all resource attributes which have been marked as auditable with the `audit` metaparameter. (The most recent cached catalog is used to determine which resource attributes are auditable.)

Inspection reports are handled identically to standard reports, and must be differentiated at parse time by your report tools; see the report format documentation for more details. Although a future version of Puppet Dashboard will support viewing of inspection reports, Puppet Labs does not currently ship any inspection report tools.

Puppet inspect was added in Puppet 2.6.5.

facter

Puppet agent nodes use a library (and associated front-end tool) called facter to provide information about the hardware and OS (version information, IP address, etc) to the puppet master server. These facts are exposed to Puppet manifests as global variables, which can be used in conditionals, string expressions, and templates. To see a list of the facts any node offers, simply open a shell session on that node and run facter. Facter is included with (and required by) all Puppet installations.

Chapter 9

Introduction to Puppet

Why Puppet

As system administrators acquire more and more systems to manage, automation of mundane tasks is increasingly important. Rather than develop in-house scripts, it is desirable to share a system that everyone can use, and invest in tools that can be used regardless of one's employer. Certainly doing things manually doesn't scale.

Puppet has been developed to help the sysadmin community move to building and sharing mature tools that avoid the duplication of everyone solving the same problem. It does so in two ways:

- It provides a powerful framework to simplify the majority of the technical tasks that sysadmins need to perform
- The sysadmin work is written as code in Puppet's custom language which is shareable just like any other code.

This means that your work as a sysadmin can get done much faster, because you can have Puppet handle most or all of the details, and you can download code from other sysadmins to help you get done even faster. The majority of Puppet implementations use at least one or two modules developed by someone else, and there are already hundreds of modules developed and shared by the community.

Learning Recommendations

We're glad you want to learn Puppet. You're free to browse around the documentation as you like, though we generally recommend trying out Puppet locally first (without the daemon and client/server setup), so you can understand the basic concepts. From there, move on to centrally managed server infrastructure. Ralsh is also a great way to get your feet wet exploring the Puppet model, after you have read some of the basic information — you can quickly see how the declarative model works for simple things like users, services, and file permissions.

Once you've learned the basics, make sure you understand classes and modules, then move on to the advanced sections and read more about the features that are useful to you. Learning all at once is definitely not required. If you find something confusing, use the feedback tab to let us know.

System Components

Puppet is typically (but not always) used in a client/server formation, with all of your clients talking to one or more central servers. Each client contacts the server periodically (every half hour, by default), downloads the latest configuration, and makes sure it is in sync with that configuration. Once done, the client can send a report back to the server indicating if anything needed to change. This diagram shows the data flow in a regular Puppet implementation:

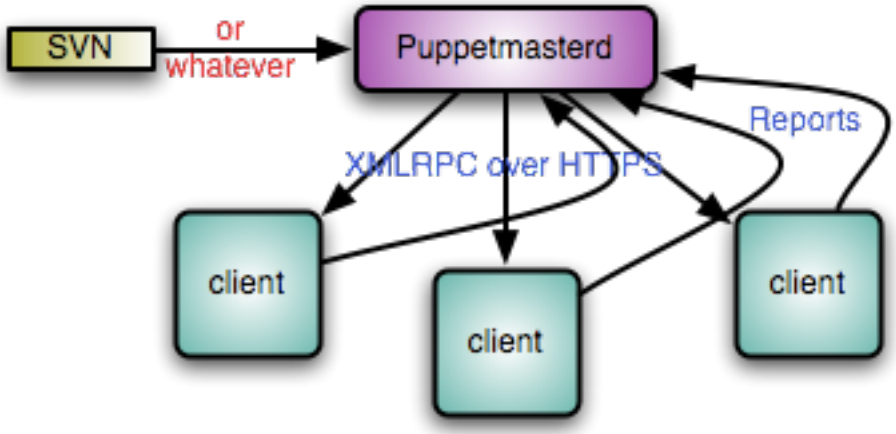

Figure 9.1: image

Puppet's functionality is built as a stack of separate layers, each responsible for a fixed aspect of the system, with tight controls on how information passes between layers:

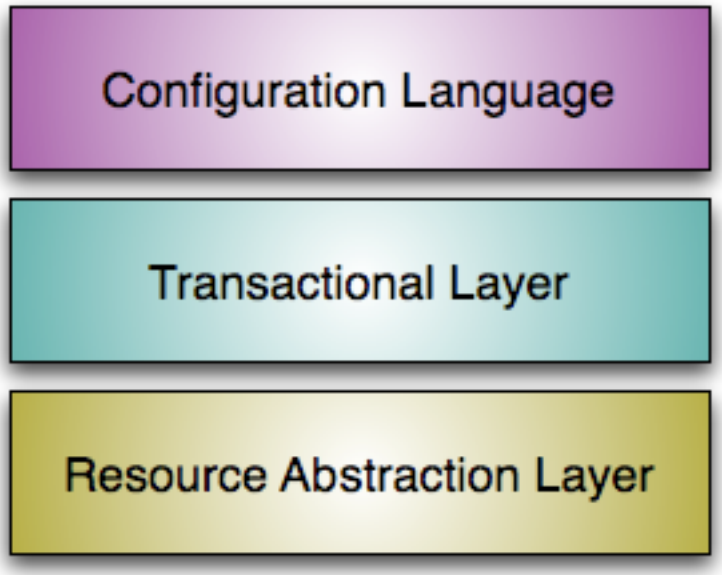

Figure 9.2: image

See also Configuring Puppet. For more information about components (puppetmasterd, puppetd, puppet, and so on), see the Tools section.

Features of the System

Idempotency

One big difference between Puppet and most other tools is that Puppet configurations are idempotent, meaning they can safely be run multiple times. Once you develop your configuration, your machines will apply the configuration often — by default, every 30 minutes — and Puppet will only make any changes to the system if the system state does not match the configured state.

If you tell the system to operate in no-op ("aka dry-run"), mode, using the `--noop` argument to one of the Puppet tools, puppet will guarantee that no work happens on your system. Similarly, if any changes do happen when running without that flag, puppet will ensure those changes are logged.

Because of this, you can use Puppet to manage a machine throughout its lifecycle — from initial installation, to ongoing upgrades, and finally to end-of-life, where you move services elsewhere. Unlike system install tools like Sun's Jumpstart or Red Hat's Kickstart, Puppet configurations can keep machines up to date for years, rather than just building them correctly only the first time and then neccessitating a rebuild. Puppet users usually do just enough with their host install tools to boostrap Puppet, then they use Puppet to do everything else.

Cross Platform

Puppet's Resource Abstraction Layer (RAL) allows you to focus on the parts of the system you care about, ignoring implementation details like command names, arguments, and file formats — your tools should treat all users the same, whether the user is stored in NetInfo or `/etc/passwd`. We call these system entities `resources`.

Ralsh, listed in the Tools section is a fun way to try out the RAL before you get too deep into Puppet language.

Model & Graph Based

Resource Types

The concept of each resource (like service, file, user, group, etc) is modelled as a "type". Puppet decouples the definition from how that implementation is fulfilled on a particular operating system, for instance, a Linux user versus an OS X user can be talked about in the same way but are implemented differently inside of Puppet.

See Type Guides for a list of managed types and information about how to use them.

Providers

Providers are the fulfillment of a resource. For instance, for the package type, both 'yum' and 'apt' are valid ways to manage packages. Sometimes more than one provider will be available on a particular platform, though each platform always has a default provider. There are currently 17 providers for the package type.

Modifying the System

Puppet resource providers are what are responsible for directly managing the bits on disk. You do not directly modify a system from Puppet language — you use the language to specify a resource, which then modifies the system. This way puppet language behaves exactly the same way in a centrally managed server setup as it does locally without a server. Rather than tacking a couple of lines onto the end of your `fstab`, you use the `mount` type to create a new resource that knows how to modify the `fstab`, or NetInfo, or wherever mount information is kept.

Resources have attributes called 'properties' which change the way a resource is managed. For instance, users have an attribute that specicies whether the home directory should be created.

'Metaparams' are another special kind of attribute, those exist on all resources. This include things like the log level for the resource, whether the resource should be in `noop` mode so it never modifies the system, and the relationships between resources.

Resource Relationships

Puppet has a system of modelling relationships between resources — what resources should be evaluated before or after one another. They also are used to determine whether a resource needs to respond to changes in another

resource (such as if a service needs to restart if the configuration file for the service has changed). This ordering reduces unneccessary commands, such as avoiding restarting a service if the configuration has *not* changed.

Because the system is graph based, it's actually possible to generate a diagram (from Puppet) of the relationships between all of your resources.

Learning The Language

Seeing a few examples in action will greatly help in learning the system.

For information about the Puppet language, see the excellent language guide

Chapter 10

Supported Platforms

Learn what platforms are supported.

Please contact Puppet Labs if you are interested in a platform not on this list.

Puppet requires Ruby to run and currently supports Ruby version 1.8.1 to 1.8.7. Ruby 1.9.x is not yet supported.

Linux

- CentOS
- Debian 3.1 and later
- Fedora Core 2–6
- Fedora 7 and later
- Gentoo Linux
- Mandriva Corporate Server 4
- RHEL 3 and later
- Oracle Linux
- SuSE Linux 8 and later
- Ubuntu 7.04 and later
- ArchLinux

BSD

- FreeBSD 4.7 and later
- OpenBSD 4.1 and later

Other Unix

- Macintosh OS X
- Sun Solaris 2.6
- Sun Solaris 7 and later
- AIX
- HP-UX

Windows

- Windows (version 2.6.0 and later)

Chapter 11

Installation Guide

This guide covers in-depth installation instructions and options for Puppet on a wide-range of operating systems.

Before Starting

You will need to install Puppet on all machines on both clients and the central Puppet master server(s).

For most platforms, you can install 'puppet' via your package manager of choice. For a few platforms, you will need to install using the tarball or RubyGems.

INFO: For instructions on installing puppet using a distribution-specific package manager, consult your operating system documentation. Volunteer contributed operating system packages can also be found on the downloads page

Ruby Prerequisites

The only prerequisite for Puppet that doesn't come as part of the Ruby standard library is facter, which is also developed by Puppet Labs.

All other prerequisites Ruby libraries should come with any standard Ruby 1.8.2+ install. Should your OS not come with the complete standard library (or you are using a custom Ruby build), these include:

- base64
- cgi
- digest/md5
- etc
- fileutils
- ipaddr
- openssl
- strscan
- syslog
- uri
- webrick
- webrick/https

- xmlrpc

NOTE: We strongly recommend using the version of Ruby that comes with your system, since that will have a higher degree of testing coverage. If you feel the particular need to build Ruby manually, you can get the source from ruby-lang.org.

OS Packages

If installing from a distribution maintained package, such as those listed on the Downloading Puppet Wiki Page all OS prerequisites should be handled by your package manager. See the Wiki for information on how to enable repositories for your particular OS. Usually the latest stable version is available as a package. If you would like to do puppet-development or see the latest versions, however, you will want to install from source.

Installing Facter From Source

The facter library is a prerequisite for Puppet. Like Puppet, there are packages available for most platforms, though you may want to use the tarball if you would like to try a newer version or are using a platform without an OS package:

Get the latest tarball:

```
$ wget http://puppetlabs.com/downloads/facter/facter-latest.tgz
```

Untar and install facter:

```
$ gzip -d -c facter-latest.tgz | tar xf -
$ cd facter-*
$ sudo ruby install.rb # or become root and run install.rb
```

There are also gems available in the download directory.

Installing Puppet From Source

Using the same mechanism as Facter, install the puppet libraries and executables:

```
# get the latest tarball
$ wget http://puppetlabs.com/downloads/puppet/puppet-latest.tgz
# untar and install it
$ gzip -d -c puppet-latest.tgz | tar xf -
$ cd puppet-*
$ sudo ruby install.rb # or become root and run install.rb
```

You can also check the source out from the git repo:

```
$ mkdir -p ~/git && cd ~/git
$ git clone git://github.com/puppetlabs/puppet
$ cd puppet
$ sudo ruby ./install.rb
```

To install into a different location you can use:

```
$ sudo ruby install.rb --bindir=/usr/bin --sbindir=/usr/sbin
```

Alternative Install Method: Using Ruby Gems

You can also install Facter and Puppet via gems:

```
$ wget http://puppetlabs.com/downloads/gems/facter-1.5.7.gem
$ sudo gem install facter-1.5.7.gem
$ wget http://puppetlabs.com/downloads/gems/puppet-0.25.1.gem
$ sudo gem install puppet-0.25.1.gem
```

Find the latest gems here

For more information on Ruby Gems, see the Gems User Guide

WARNING: If you get the error, in `require: no such file to load`, define the RUBYOPT environment variable as advised in the post-install instructions of the RubyGems User Guide.

Configuring Puppet

Now that the packages are installed, see Configuring Puppet for setup instructions.

Chapter 12

Configuring Puppet

Puppet's behavior can be customized with a rather large collection of settings. Most of these can be safely ignored, but you'll almost definitely have to modify some of them.

This document describes how Puppet's configuration settings work, and describes all of Puppet's auxiliary config files.

Puppet's Settings

Puppet is able to automatically generate a reference of all its config settings (puppet doc --reference configuration), and the documentation site includes archived references for every recent version of Puppet. You will generally want to consult the the most recent stable version's reference.

When retrieving the value for a given setting, Puppet follows a simple lookup path, stopping at the first value it finds. In order, it will check:

- Values specified on the command line
- Values in environment blocks in puppet.conf
- Values in run mode blocks in puppet.conf
- Values in the main block of puppet.conf
- The default values

The settings you'll have to interact with will vary a lot, depending on what you're doing with Puppet. But at the least, you should get familiar with the following:

- `certname` — The locally unique name for this node. If you aren't using DNS names to identify your nodes, you'll need to set it yourself.
- `server` — The puppet master server to request configurations from.
- `confdir` — One of Puppet's main working directories, which usually contains config files, manifests, modules, and certificates.
- `vardir` — Puppet's other main working directory, which usually contains cached data and configurations, reports, and file backups.
- `modulepath` — The search path for Puppet modules.
- `environment` — On agent nodes, the environment to request configuration in.

- `node_terminus` — How puppet master should get node definitions; if you use an ENC, you'll need to set it to "exec."
- `external_nodes` — The script to run for node definitions (if you chose a node_terminus of "exec").
- `report` — Whether to send reports to the puppet master.
- `reports` — On the puppet master, which report handler(s) to use.

`puppet.conf`

Puppet's main config file is puppet.conf, which is located in Puppet's confdir. Under Puppet Enterprise, the confdir is /etc/puppetlabs/puppet; on most other systems, the confdir is /etc/puppet when running as root or the Puppet user and ~/.puppet when running as a normal user.

File Format

puppet.conf uses an INI-like format, with [config blocks] containing indented groups of setting = value lines. Comment lines # start with an octothorpe; partial-line comments are not allowed.

You can interpolate the value of a setting by using its name as a $variable. (Note that $environment has special behavior: most of the Puppet applications will interpolate their own environment, but puppet master will use the environment of the agent node it is serving.)

If a setting has multiple values, they should be a comma-separated list. "Path"-type settings made up of multiple directories should use the system path separator (colon, on most Unices).

Finally, for settings that accept only a single file or directory, you can set the owner, group, and/or mode by putting their desired states in curly braces after the value.

Putting that all together:

```
# a block:
[main]
  # setting = value pairs:
  server = master.puppetlabs.lan
  certname = 005056c00008.localcloud.puppetlabs.lan

  # variable interpolation:
  rundir = $vardir/run
  modulepath = /etc/puppet/modules/$environment:/usr/share/puppet/modules
[master]
  # a list:
  reports = store, http

  # a multi-directory modulepath:
  modulepath = /etc/puppet/modules:/usr/share/puppet/modules

  # setting owner and mode for a directory:
  vardir = /Volumes/zfs/vardir {owner = puppet, mode = 644}
```

Config Blocks

Settings in different config blocks take effect under varying conditions. Settings in a more specific block can override those in a less specific block, as per the lookup path described above.

The `[main]` Block

The [main] config block is the least specific. Settings here are always effective, unless overridden by a more specific block.

`[agent]`, `[master]`, and `[user]` Blocks

These three blocks correspond to Puppet's run modes. Settings in [agent] will only be used by puppet agent, settings in [master] will be used by puppet master and puppet cert, and settings in [user] will be used by puppet apply. The Faces subcommands introduced in Puppet 2.7 default to the user run mode, but their mode can be changed at run time with the --mode option. Note that not every setting makes sense for every run mode, but specifying a setting in a block where it is irrelevant has no observable effect.

Notes on Puppet 0.25.5 and Older Prior to Puppet 2.6, blocks were assigned by application name rather than by run mode; e.g. [puppetd], [puppetmasterd], [puppet], and [puppetca]. Although these names still work, their use is deprecated, and they interact poorly with the modern run mode blocks. If you have an older config file and are using Puppet 2.6 or later, you should consider changing [puppetd] to [agent], [puppet] to [user], and combining [puppetmasterd] and [puppetca] into [master].

Per-environment Blocks

Blocks named for environments are the most specific, and can override settings in the run mode blocks. Only a small number of settings (specifically: modulepath, manifest, manifestdir, and templatedir) can be set in a per-environment block; any other settings will be ignored and read from a run mode or main block.

Like with the $environment variable, puppet master treats environments differently from the other run modes: instead of using the block corresponding to its own environment setting, it will use the block corresponding to each agent node's environment. The puppet master's own environment setting is effectively inert.

Command-Line Options

You can override any config setting at runtime by specifying it as a command-line option to almost any Puppet application. (Puppet doc is the main exception.)

Boolean settings are handled a little differently: use a bare option for a true value, and add a prefix of no– for false:

```
# Equivalent to listen = true:
$ puppet agent --listen
# Equivalent to listen = false:
$ puppet agent --no-listen
```

For non-boolean settings, just follow the option with the desired value:

```
$ puppet agent --certname magpie.puppetlabs.lan
# An equals sign is optional:
$ puppet agent --certname=magpie.puppetlabs.lan
```

Inspecting Settings

Puppet agent, apply, and master all accept the --configprint <setting> option, which makes them print their local value of the requested setting and exit. In Puppet 2.7, you can also use the puppet config print <setting> action, and view values in different run modes with the --mode flag. Either way, you can view all settings by passing all instead of a specific setting.

```
$ puppet master --configprint modulepath
# or:
$ puppet config print modulepath --mode master

/etc/puppet/modules:/usr/share/puppet/modules
```

Puppet agent, apply, and master also accept a --genconfig option, which behaves similarly to --configprint all but outputs a complete puppet.conf file, with descriptive comments for each setting, default values explicitly declared, and settings irrelevant to the requested run mode commented out. Having the documentation inline and the default values laid out explicitly can be helpful for setting up your config file, or it can be noisy and hard to work with; it comes down to personal taste.

You can also inspect settings for specific environments with the --environment option:

```
$ puppet agent --environment testing --configprint modulepath
/etc/puppet/testing/modules:/usr/share/puppet/modules
```

(As implied above, this doesn't work in the master run mode, since the master effectively has no environment.)

Other configuration files

In addition to the main configuration file, there are five special-purpose config files you might need to interact with: auth.conf, fileserver.conf, tagmail.conf, autosign.conf, and device.conf.

`auth.conf`

Access to Puppet's REST API is configured in auth.conf, the location of which is determined by the rest_authconfig setting. (Default: /etc/puppet/auth.conf.) It consists of a series of ACL stanzas, and behaves quite differently from puppet.conf; for full details, see the REST access control documentation.

```
# Example auth.conf:

path /
auth any
environment override
allow magpie.lan

path /certificate_status
auth any
environment production
allow magpie.lan

path /facts
method save
auth any
allow magpie.lan

path /facts
auth yes
method find, search
allow magpie.lan, dashboard, redmaster.magpie.lan
```

`fileserver.conf`

By default, fileserver.conf isn't necessary, provided that you only need to serve files from modules. If you want to create additional fileserver mount points, you can do so in /etc/puppet/fileserver.conf (or whatever is set in the fileserverconfig setting).

fileserver.conf consists of a collection of mount-point stanzas, and looks like a hybrid of puppet.conf and auth.conf:

```
# Files in the /path/to/files directory will be served
# at puppet:///mount_point/.
[mount_point]
    path /path/to/files
    allow *.domain.com
    deny *.wireless.domain.com
```

See the file serving documentation for more details.

Note that certname globs do not function as normal globs: an asterisk can only represent one or more subdomains at the front of a certname that resembles a fully-qualified domain name. (That is, if your certnames don't look like FQDNs, you can't use autosign.conf to full effect.

`tagmail.conf`

Your puppet master server can send targeted emails to different admin users whenever certain resources are changed. This requires that you:

- Set report = true on your agent nodes

- Set reports = tagmail on the puppet master (reports accepts a list, so you can enable any number of reports)
- Set the reportfrom email address and either the smtpserver or sendmail setting on the puppet master
- Create a tagmail.conf file at the location specified in the tagmap setting

More details are available at the tagmail report reference.

The tagmail.conf file is list of lines, each of which consists of a tag, a colon, and an email address. The tag portion of a line can also be a !negated tag, a list of tags, or the word "all," which does exactly what it sounds like.

```
all: zach@puppetlabs.com
webserver, !mailserver: httpadmins@domain.com
```

autosign.conf

The autosign.conf file (located at /etc/puppet/autosign.conf by default, and configurable with the autosign setting) is a list of certnames or certname globs (one per line) whose certificate requests will automatically be signed.

```
rebuilt.puppetlabs.lan
*.magpie.puppetlabs.lan
*.local
```

Note that certname globs do not function as normal globs: an asterisk can only represent one or more subdomains at the front of a certname that resembles a fully-qualified domain name. (That is, if your certnames don't look like FQDNs, you can't use autosign.conf to full effect.

As any host can provide any certname, autosigning should only be used with great care, and only in situations where you essentially trust any computer able to connect to the puppet master.

device.conf

Puppet device, added in Puppet 2.7, configures network hardware using a catalog downloaded from the puppet master; in order to function, it requires that the relevant devices be configured in /etc/puppet/device.conf (configurable with the deviceconfig setting).

device.conf is organized in INI-like blocks, with one block per device:

```
[device certname]
    type <type>
    url <url>
[router6.puppetlabs.lan]
    type cisco
    url ssh://admin:password@ef03c87a.local
```

Chapter 13

Scaling Puppet

Tune Puppet for maximum performance in large environments.

Are you using the default webserver?

WEBrick, the default web server used to enable Puppet's web services connectivity, is essentially a reference implementation, and becomes unreliable beyond about ten managed nodes. In any sort of production environment, you should switch to a more efficient web server implementation such as Passenger or Mongrel, which will allow for serving many more nodes concurrently. If your system can work with Passenger, that is currently the recommended route. On older systems, use Mongrel.

Delayed check in

Puppet's default configuration asks that each node check in every 30 minutes. An option called 'splay' can add a random configurable lag to this check in time, to further balance out check in frequency. Alternatively, do not run puppetd as a daemon, and add puppet agent with −−onetime to your crontab, allowing for setting different crontab intervals on different servers.

Triggered selective updates

Similar to the delayed checkin and cron strategies, it's possible to trigger node updates on an as-needed basis. Managed nodes can be configured to not check in automatically every 30 minutes, but rather to check in only when requested. puppetrun (in the 'ext' directory of the Puppet checkout) may be used to selectively update hosts. Alternatively, do not run the daemon, and a tool like mcollective could be used to launch puppet agent with the −−onetime option.

No central host

Using a central server offers numerous advantages, particularly in the area of security and enhanced control. In environments that do not need these features, it is possible to use rsync, git, or some other means to transfer Puppet manifests and data to each individual node, and then run puppet apply locally (usually via cron). This approach scales essentially infinitely, and full usage of Puppet and facter is still possible.

Minimize recursive file serving

Puppet's recursive file serving works well for small directories, but it isn't as efficient as rsync or NFS, and using it for larger directories can take a performance toll on both the client and server.

Chapter 14

Passenger

Using Passenger instead of WEBrick for web services offers numerous performance advantages. This guide shows how to set it up.

Supported Versions

Passenger support is present in release 0.24.6 and later versions only. For earlier versions, consider Using Mongrel.

Why Passenger

Traditionally, the puppetmaster would embed a WEBrick or Mongrel Web Server to serve the puppet clients. This may work well for you, but a few people feel like using a proven web server like Apache would be superior for this purpose.

What is Passenger?

Passenger (AKA mod_rails or mod_rack) is the Apache 2.x Extension which lets you run Rails or Rack applications inside Apache.

Puppet (>0.24.6) now ships with a Rack application which can embed a puppetmaster. While it should be compatible with every Rack application server, it has only been tested with Passenger.

Depending on your operating system, the versions of Puppet, Apache and Passenger may not support this implementation. Specifically, Ubuntu Hardy ships with an older version of puppet (0.24.4) and doesn't include passenger at all, however updated packages for puppet can be found here. There are also some passenger packages there, but as of 2009–09–28 they do not seem to have the latest passenger (2.2.5), so better install passenger from a gem as per the instructions at [modrails.com].

Note: Passenger versions 2.2.3 and 2.2.4 have known bugs regarding to the SSL environment variables, which make them unsuitable for hosting a puppetmaster. So use either 2.2.2, or 2.2.5. Note that while it was expected that Passenger 2.2.2 would be the last version which can host a 0.24.x puppetmaster, that turns out to be not true, cf. this bug report. So, passenger 2.2.5 works fine.

Installation Instructions for Puppet 0.25.x and 2.6.x

Please see ext/rack/README in the puppet source tree for instructions.

Whatever you do, make sure your config.ru file is owned by the puppet user! Passenger will setuid to that user.

Installation Instructions for Puppet 0.24.x for Debian/Ubuntu and RHEL5

Make sure puppetmasterd ran at least once, so puppetmasterd SSL certificates are setup intially.

Install Apache 2, Rack and Passenger

For Debian/Ubuntu:

```
apt-get install apache2
apt-get install ruby1.8-dev
```

For RHEL5 (needs the EPEL repository enabled):

```
yum install httpd httpd-devel ruby-devel rubygems
```

Install Rack/Passenger

The latest version of Passenger (2.2.5) appears to work fine on RHEL5:

```
gem install rack
gem install passenger
passenger-install-apache2-module
```

If you want the older 2.2.2 gem, you could manually download the .gem file from RubyForge. Or, you could just add the correct versions to your gem command:

```
  gem install -v 0.4.0 rack
  gem install -v 2.2.2 passenger
```

Enable Apache modules "ssl" and "headers":

```
# for Debian or Ubuntu:
a2enmod ssl
a2enmod headers

# for RHEL5
yum install mod_ssl
```

Configure Apache

For Debian/Ubuntu:

```
cp apache2.conf /etc/apache2/sites-available/puppetmasterd  (see below for the file contents)
ln -s /etc/apache2/sites-available/puppetmasterd /etc/apache2/sites-enabled/puppetmasterd
vim /etc/apache2/conf.d/puppetmasterd (replace the hostnames)
```

For RHEL5:

```
cp puppetmaster.conf /etc/httpd/conf.d/ (see below for file contents)
vim /etc/httpd/conf.d/puppetmaster.conf (replace hostnames with corrent values)
```

Install the rack application [1]:

```
mkdir -p /usr/share/puppet/rack/puppetmasterd
mkdir /usr/share/puppet/rack/puppetmasterd/public /usr/share/puppet/rack/puppetmasterd/tmp
cp config.ru /usr/share/puppet/rack/puppetmasterd
chown puppet /usr/share/puppet/rack/puppetmasterd/config.ru
```

Go:

```
# For Debian/Ubuntu
/etc/init.d/apache2 restart

# For RHEL5
/etc/init.d/httpd restart
```

If all works well, you'll want to make sure your puppmetmasterd init script does not get called anymore:

```
# For Debian/Ubuntu
update-rc.d -f puppetmaster remove

# For RHEL5
chkconfig puppetmaster off
chkconfig httpd on
```

[1] Passenger will not let applications run as root or the Apache user, instead an implicit setuid will be done, to the user whom owns config.ru. Therefore, config.ru shall be owned by the puppet user.

Apache Configuration for Puppet 0.24.x

This Apache Virtual Host configures the puppetmaster on the default puppetmaster port (8140).

```
Listen 8140
<VirtualHost *:8140>

    SSLEngine on
    SSLCipherSuite SSLv2:-LOW:-EXPORT:RC4+RSA
    SSLCertificateFile      /var/lib/puppet/ssl/certs/puppet-server.inqnet.at.pem
    SSLCertificateKeyFile   /var/lib/puppet/ssl/private_keys/puppet-server.inqnet.at.pem
    SSLCertificateChainFile /var/lib/puppet/ssl/ca/ca_crt.pem
    SSLCACertificateFile    /var/lib/puppet/ssl/ca/ca_crt.pem
    # CRL checking should be enabled; if you have problems with Apache complaining about the CRL, disable
        the next line
    SSLCARevocationFile     /var/lib/puppet/ssl/ca/ca_crl.pem
    SSLVerifyClient optional
    SSLVerifyDepth  1
    SSLOptions +StdEnvVars

    # The following client headers allow the same configuration to work with Pound.
    RequestHeader set X-SSL-Subject %{SSL_CLIENT_S_DN}e
    RequestHeader set X-Client-DN %{SSL_CLIENT_S_DN}e
    RequestHeader set X-Client-Verify %{SSL_CLIENT_VERIFY}e

    RackAutoDetect On
    DocumentRoot /usr/share/puppet/rack/puppetmasterd/public/
    <Directory /usr/share/puppet/rack/puppetmasterd/>
        Options None
        AllowOverride None
        Order allow,deny
        allow from all
    </Directory>
</VirtualHost>
```

If the current puppetmaster is not a certificate authority, you may need to change the following lines. The certs/ca.pem file should exist as long as the puppetmaster has been signed by the CA.

```
  SSLCertificateChainFile /var/lib/puppet/ssl/certs/ca.pem
    SSLCACertificateFile    /var/lib/puppet/ssl/certs/ca.pem
```

For Debian hosts you might wish to add:

```
  LoadModule passenger_module /var/lib/gems/1.8/gems/passenger-2.2.5/ext/apache2/mod_passenger.so
    PassengerRoot /var/lib/gems/1.8/gems/passenger-2.2.5
    PassengerRuby /usr/bin/ruby1.8
```

For RHEL hosts you may need to add:

```
   LoadModule passenger_module /usr/lib/ruby/gems/1.8/gems/passenger-2.2.5/ext/apache2/mod_passenger.so
   PassengerRoot /usr/lib/ruby/gems/1.8/gems/passenger-2.2.5
   PassengerRuby /usr/bin/ruby
```

For details about enabling and configuring Passenger, see the Passenger install guide.

The config.ru file for Puppet 0.24.x

```
# This file is mostly based on puppetmasterd, which is part of
# the standard puppet distribution.

require 'rack'
require 'puppet'
require 'puppet/network/http_server/rack'

# startup code stolen from bin/puppetmasterd
Puppet.parse_config
Puppet::Util::Log.level = :info
Puppet::Util::Log.newdestination(:syslog)
# A temporary solution, to at least make the master work for now.
Puppet::Node::Facts.terminus_class = :yaml
# Cache our nodes in yaml.  Currently not configurable.
Puppet::Node.cache_class = :yaml


# The list of handlers running inside this puppetmaster
handlers = {
    :Status => {},
    :FileServer => {},
    :Master => {},
    :CA => {},
    :FileBucket => {},
    :Report => {}
}

# Fire up the Rack-Server instance
server = Puppet::Network::HTTPServer::Rack.new(handlers)

# prepare the rack app
app = proc do |env|
    server.process(env)
end

# Go.
run app
```

If you don't want to run with the CA enabled, you could drop the ':CA => {}' line from the config.ru above.

The config.ru file for 0.25.x

Please see ext/rack in the 0.25 source tree for the proper config.ru file.

Suggested Tweaks

Larry Ludwig's testing of passenger/puppetmasterd recommends adjusting these options in your apache configuration:

- PassengerPoolIdleTime 300 - Set to 5 min (300 seconds) or less. The shorting this option allows for puppetmasterd to get refreshed at some interval. This option is also somewhat dependent upon the amount of puppetd nodes connecting and at what interval.
- PassengerMaxPoolSize 15 - to 15% more instances than what's needed. This will allow idle puppetmasterd to get recycled. The net effect is less memory will be used, not more.
- PassengerUseGlobalQueue on - Since communication with the puppetmaster from puppetd is a long process (more than 20 seconds in most cases) and will allow for processes to get recycled better
- PassengerHighPerformance on - The additional Passenger features for apache compatibility are not needed with Puppet.

As is expected with traditional web servers, once your service starts using swap, performance degradation will occur — so be mindful of your memory/swap usage on your Puppetmaster.

To monitor the age of your puppetmasterd processes within Passenger, run

```
passenger-status | grep PID | sort
```

```
  PID: 14590    Sessions: 1     Processed: 458      Uptime: 3m 40s
  PID: 7117     Sessions: 0     Processed: 10980    Uptime: 1h 43m 41s
  PID: 7355     Sessions: 0     Processed: 9736     Uptime: 1h 38m 38s
  PID: 7575     Sessions: 0     Processed: 9395     Uptime: 1h 32m 27s
  PID: 9950     Sessions: 0     Processed: 6581     Uptime: 1h 2m 35s
```

Passenger can be configured to be recycling puppetmasterd every few hours to ensure memory/garbage collection from Ruby is not a factor.

Chapter 15

Using Mongrel

Puppet daemons default to using WEBrick for http serving, but puppetmasterd can be used with Mongrel instead for performance benefits.

The mongrel documentation is currently maintained our our Wiki until it can be migrated over. Please see the OS specific setup documents on the Wiki for further information.

Chapter 16

Techniques

Here are some useful tips & tricks.

How Can I Manage Whole Directories of Files Without Explicitly Listing the Files?

The file type has a "recurse" attribute, which can be used to synchronize the contents of a target directory recursively with a chosen source. In the example below, the entire /etc/httpd/conf.d directory is synchronized recursively with the copy on the server:

```
file { "/etc/httpd/conf.d":
  source => "puppet://server/vol/mnt1/adm/httpd/conf.d",
  recurse => true,
}
```

You can also set `purge => true` to keep the directory clear of all files or directories not managed by Puppet.

How Do I Run a Command Whenever A File Changes?

The answer is to use an exec resource with refreshonly set to true, such as in this case of telling bind to reload its configuration when it changes:

```
file { "/etc/bind": source => "/dist/apps/bind" }

exec { "/usr/bin/ndc reload":
  subscribe => File["/etc/bind"],
  refreshonly => true
}
```

The exec has to subscribe to the file so it gets notified of changes.

How Can I Ensure a Group Exists Before Creating a User?

In the example given below, we'd like to create a user called tim who we want to put in the fearme group. By using the require attribute, we can create a dependency between the user tim and the group fearme. The result is that user tim will not be created until puppet is certain that the fearme group exists.

```
group { "fearme":
        ensure => present,
        gid => 1000
}
user { "tim":
        ensure => present,
```

```
        gid => "fearme",
        groups => ["adm", "staff", "root"],
        membership => minimum,
        shell => "/bin/bash",
        require => Group["fearme"]
}
```

Note that Puppet will set this relationship up for you automatically, so you should not normally need to do this.

How Can I Require Multiple Resources Simultaneously?

Give the require attribute an array as its value. In the example given below, we're again adding the user tim (just as we did earlier in this document), but in addition to requiring tim's primary group, fearme, we're also requiring another group, fearmenot. Any reasonable number of resources can be required in this way.

```
user { "tim":
        ensure => present,
        gid => "fearme",
        groups => ["adm", "staff", "root", "fearmenot"],
        membership => minimum,
        shell => "/bin/bash",
        require => [ Group["fearme"],
                     Group["fearmenot"]
                   ]
        }
```

Can I use complex comparisons in if statements and variables?

In Puppet version 0.24.6 onwards you can use complex expressions in if statements and variable assignments. You can see examples of how to do this in the language guide.

Can I output Facter facts in YAML?

Facter supports output of facts in YAML as well as to standard out. You need to run:

```
# facter —yaml
```

To get this output, which you can redirect to a file for further processing.

Can I check the syntax of my templates?

ERB files are easy to syntax check. For a file mytemplate.erb, run:

```
$ erb -x -T '-' -P mytemplate.erb | ruby -c
```

The trim option specified corresponds to what Puppet uses.

Chapter 17

Troubleshooting

Answers to some common problems that may come up.

Basic workflow items are covered in the main section of the documentation. If you're looking for how to do something unconventional, you may also wish to read Techniques.

General

Why hasn't my new node configuration been noticed?

If you're using separate node definition files and import them into site.pp (with an `import *.node`, for example) you'll find that new files added won't get noticed until you restart puppetmasterd. This is due to the fact globs aren't evaluated on each run, but only when the 'parent' file is re-read.

To make sure your new file is actually read, simply 'touch' the site.pp (or importing file) and the glob will be re-evaluated.

Why don't my certificates show as waiting to be signed on my server when I do a "`puppet cert --list`"?

puppet cert must be run with root privileges. If you are not root, then re-run the command with sudo:

```
sudo puppet cert --list
```

I keep getting "certificates were not trusted". What's wrong?

Firstly, if you're re-installing a machine, you probably haven't cleared the previous certificate for that machine. To correct the problem:

1. Run `sudo puppet cert --clean {node certname}` on the puppet master to clear the certificates.
2. Remove the entire SSL directory of the client machine (`sudo rm -r etc/puppet/ssl; rm -r /var/lib/puppet/ssl`).

Assuming that you're not re-installing, by far the most common cause of SSL problems is that the clock on the client machine is set incorrectly, which confuses SSL because the "validFrom" date in the certificate is in the future.

You can figure the problem out by manually verifying the certificate with openssl:

```
sudo openssl verify -CAfile /etc/puppet/ssl/certs/ca.pem /etc/puppet/ssl/certs/myhostname.domain.com.pem
```

This can also happen if you've followed the Using Mongrel pattern to alleviate file download problems. If your set-up is such that the host name differs from the name in the Puppet server certificate, or there is any other SSL certificate negotiation problem, the SSL handshake between client and server will fail. In this case, either alleviate the SSL handshake problems (debug using cURL), or revert to the original Webrick installation.

I'm getting IPv6 errors; what's wrong?

This can happen if Ruby is not compiled with IPv6 support. The only known solution is to make sure you're running a version of Ruby compiled with IPv6 support.

I'm getting tlsv1 alert unknown ca errors; what's wrong?

This problem is caused by puppetmasterd not being able to read its ca certificate. This problem might occur up to 0.18.4 but has been fixed in 0.19.0. You can probably fix it for versions before 0.19.0 by changing the group ownership of the /etc/puppet/ssl directory to the puppet group, but puppetd may change the group back. Having puppetmasterd start as the root user should fix the problem permanently until you can upgrade.

Why does Puppet keep trying to start a running service?

The ideal way to check for a service is to use the hasstatus attribute, which calls the init script with its status command. This should report back to Puppet whether the service is running or stopped.

In some broken scripts, however, the status output will be correct ("Ok" or "not running"), but the exit code of the script will be incorrect. (Most commonly, the script will always blindly return 0.) Puppet only uses the exit code, and so may behave unpredictably in these cases.

There are two workarounds, and one fix. If you must deal with the script's broken behavior as is, your resource can either use the "pattern" attribute to look for a particular name in the process table, or use the "status" attribute to specify a custom script that returns the proper exit code for the service's status.

The longer-term fix is to rewrite the service's init script to use the proper exit codes. When rewriting them, or submitting bug reports to vendors or upstream, be sure to reference the LSB Init Script Actions standard. This should carry more weight by pointing out an official, published standard they're failing to meet, rather than trying to explain how their bug is causing problems in Puppet.

Why is my external node configuration failing? I get no errors by running the script by hand.

Most of the time, if you get the following error when running you client

```
warning: Not using cache on failed catalog
err: Could not retrieve catalog; skipping run
```

it is because of some invalid YAML output from your external node script. Check yaml.org if you have doubts about validity.

Puppet Syntax Errors

Puppet generates syntax errors when manifests are incorrectly written. Sometimes these errors can be a little cryptic. Below is a list of common errors and their explanations that should help you trouble-shoot your manifests.

Syntax error at '}'; expected '}' at manifest.pp:nnn

This error can occur when:

```
service { "fred" }
```

This contrived example demonstrates one way to get the very confusing error of Puppet's parser expecting what it found. In this example, the colon (:) is missing after the service title. A variant looks like:

```
service { "fred"
    ensure => running
}
```

and the error would be Syntax error at 'ensure'; expected '}' .

You can also get the same error if you forget a comma. For instance, in this example the comma is missing at the end of line 3: service { "myservice": provider => "runit" path => "/path/to/daemons" }

Syntax error at ':'; expected ']' at manifest.pp:nnn

This error can occur when:

```
classname::define_name {
    "jdbc/automation":
        cpoolid     => "automationPool",
        require     => [ Classname::other_define_name["automationPool"] ],
}
```

The problem here is that Puppet requires that object references in the require lines to begin with a capital letter. However, since this is a reference to a class and a define, the define also needs to have a capital letter, so Classname::Other_define_name would be the correct syntax.

Syntax error at ':'; expected '}' at manifest.pp:nnn

This error happens when you use unquoted comparators with dots in them, a'la:

```
class autofs {

  case $kernelversion {
    2.6.9:   { $autofs_packages = ["autofs", "autofs5"] }
    default: { $autofs_packages = ["autofs"] }
  }
}
```

That 2.6.9 needs to have double quotes around it, like so:

```
class autofs {

   case $kernelversion {
     "2.6.9":   { $autofs_packages = ["autofs", "autofs5"] }
     default: { $autofs_packages = ["autofs"] }
   }
 }
```

Could not match '__define_name' at manifest.pp:nnn on node nodename

This error can occur using a manifest like:

```
case $ensure {
    "present": {
        _define_name {
            "$title":
                user        => $user,
        }
    }
}
```

This one is simple - you cannot begin a function name (define name) with an underscore.

Duplicate definition: Classname::Define_name[system] is already defined in file manifest.pp at line nnn; cannot redefine at manifest.pp:nnn on node nodename

This error can occur when using a manifest like:

```
Classname::define_name {
      "system":
          properties  => "Name=system";
      .....
      "system":
           properties  => "Name=system";
 }
```

The most confusing part of this error is that the line numbers are usually the same - this is the case when using the block format that Puppet supports for a resource definition. In this contrived example, the system entry has been defined twice, so one of them needs removing.

Syntax error at '=>'; expected ')'

This error results from incorrect syntax in a defined resource type:

```
define foo($param => 'value') { ... }
```

Default values for parameters are assigned, not defined, therefore a '=', not a '=>' operator is needed.

err: Exported resource Blah[$some_title] cannot override local resource on node $nodename

While this is not a classic "syntax" error, it is a annoying error none-the-less. The actual error tells you that you have a local resource Blah[$some_title] that puppet refuses to overwrite with a collected resource of the same name. What most often happens, that the same resource is exported by two nodes. One of them is collected first and when trying to collect the second resource, this error happens as the first is already converted to a "local" resource.

Common Misconceptions

Node Inheritance and Variable Scope

It is generally assumed that the following will result in the /tmp/puppet-test.variable file containing the string 'my_node':

```
class test_class {
    file { "/tmp/puppet-test.variable":
       content => "$testname",
       ensure => present,
    }
}

node base_node {
    include test_class
}

node my_node inherits base_node {
    $testname = 'my_node'
}
```

Contrary to expectations, /tmp/puppet-test.variable is created with no contents. This is because the inherited test_class remains in the scope of base_node, where $testname is undefined.

Node inheritance is currently only really useful for inheriting static or self-contained classes, and is as a result of quite limited value.

A workaround is to define classes for your node types - essentially include classes rather than inheriting them. For example:

```
class test_class {
    file { "/tmp/puppet-test.variable":
       content => "$testname",
       ensure => present,
    }
```

```
}

class base_node_class {
    include test_class
}

node my_node {
    $testname = 'my_node'
    include base_node_class
}
```

/tmp/puppet-test.variable will now contain 'my_node' as desired.

Class Inheritance and Variable Scope

The following would also not work as generally expected:

```
class base_class {
    $myvar = 'bob'
    file {"/tmp/testvar":
        content => "$myvar",
        ensure => present,
    }
}

class child_class inherits base_class {
    $myvar = 'fred'
}
```

The /tmp/testvar file would be created with the content 'bob', as this is the value of $myvar where the type is defined.

A workaround would be to 'include' the base_class, rather than inheriting it, and also to strip the $myvar out of the included class itself (otherwise it will cause a variable scope conflict - $myvar would be set twice in the same child_class scope):

```
$myvar = 'bob'

class base_class {
    file {"/tmp/testvar":
        content => "$myvar",
        ensure => present,
    }
}

class child_class {
    $myvar = 'fred'
    include base_class
}
```

In some cases you can reset the content of the file resource so that the scope used for the content (e.g., template) is rebound. Example:

```
class base_class {
    $myvar = 'bob'
    file { "/tmp/testvar":
        content => template("john.erb"),
    }
}

class child_class inherits base_class {
    $myvar = 'fred'
    File["/tmp/testvar"] { content => template("john.erb") }
}
```

(john.erb contains a reference like <%= myvar %>.)

To avoid the duplication of the template filename, it is better to sidestep the problem altogether with a define:

```
class base_class {
    define testvar_file($myvar="bob") {
        file { $name:
            content => template("john.erb"),
        }
    }
    testvar_file { "/tmp/testvar": }
}

class child_class inherits base_class {
    Base_class::Testvar_file["/tmp/testvar"] { myvar => fred }
}
```

Whilst not directly solving the problem also useful are qualified variables that allow you to refer to variables from other classes. Qualified variables might provoke alternate methods of solving this issue. You can use qualified methods like:

```
class foo {
    $foovariable = "foobar"
}

class bar {
    $barvariable = $foo::foovariable
}
```

In this example the value of the of the $barvariable variable in the bar class will be set to foobar the value of the $foovariable variable which was set in the foo class.

Custom Type & Provider development

err: Could not retrieve catalog: Invalid parameter 'foo' for type 'bar'

When you are developing new custom types, you should restart both the puppetmasterd and the puppetd before running the configuration using the new custom type. The pluginsync feature will then synchronise the files and the new code will be loaded when both daemons are restarted.

Chapter 18

Module Organization

How to organize Puppet content inside of modules.

General Information

A Puppet module is a collection of resources, classes, files, definitions and templates. It might be used to configure Apache or a Rails module, or a Trac site or a particular Rails application.

Modules are easily re-distributable. For example, this will enable you to have the default site configuration under /etc/puppet, with modules shipped by Puppet proper in /usr/share/puppet/. You could also have other directories containing a happy mix-and-match of version control checkouts in various states of development and production readiness.

Modules are available in Puppet version 0.22.2 and later.

Configuration

There are two configuration settings that pertain to modules:

1. The search path for modules is defined with the modulepath setting in the [puppetmasterd] (pre–2.6) or [master] (post–2.6) section of the puppet master's config file, and it should be a colon-separated list of directories:

   ```
   [puppetmasterd]
   ...
   modulepath = /var/lib/puppet/modules:/data/puppet/modules
   ```

   The search path can be added to at runtime by setting the PUPPETLIB environment variable, which must also be a colon-separated list of directories.

2. Access control settings for the fileserver module [modules] are set in fileserver.conf, as described later in this page. The path configuration for that module is always ignored, and specifying a path will produce a warning.

Sources of Modules

To accommodate different locations in the file system for the different use cases, there is a configuration variable modulepath which is a list of directories to scan in turn.

A reasonable default could be configured as /etc/puppet/modules:/usr/share/puppet:/var/lib/modules. Alternatively, the /etc/puppet directory could be established as a special anonymous module which is always searched first to retain backwards compatibility to today's layout.

For some environments it might be worthwhile to consider extending the modulepath configuration item to contain branches checked out directly from version control, for example:

```
svn:file:///Volumes/svn/repos/management/master/puppet.testing/trunk
```

Naming

Module names should be restricted to lowercase alphanumeric characters and underscores, and should begin with a lowercase letter; that is, they should match the expression `[a-z][a-z0-9_]*`. Note that these are the same restrictions that apply to class names, with the added restriction that module names cannot contain the namespace separator (`::`) as modules cannot be nested.

Although some names that violate these restrictions currently work, using them is not recommended.

The module name `site` is reserved for local use and should not be used in modules meant for distribution.

Internal Organisation

A Puppet module contains manifests, distributable files, plugins and templates arranged in a specific directory structure:

```
MODULE_PATH/
downcased_module_name/
    files/
    manifests/
      init.pp
      foo.pp
    lib/
      puppet/
        parser/
          functions/
        provider/
        type/
      facter/
    templates/
    tests
      init.pp
      foo.pp
   README
```

NOTE: In Puppet versions prior to 0.25.0 the `lib` directory was named `plugins`. Other directory names are unchanged.

Each module must contain a `init.pp` manifest file at the specified location. This manifest file can contain all the classes associated with this module or additional .pp files can be added directly under the manifests folder. If adding additional .pp files, naming them after the class they define will allow auto lookup magic (explained further below in Module Lookup).

One of the things to be accomplished with modules is code sharing. A module by nature should be self-contained: one should be able to get a module from somewhere and drop it into your module path and have it work.

There are cases, however, where the module depends on generic things that most people will already have defines or classes for in their regular manifests. Instead of adding these into the manifests of your module, add them to the `depends` folder (which is basically only documenting, it doesn't change how your module works) and mention these in your README, so people can at least see exactly what your module expects from these generic dependencies, and possibly integrate them into their own regular manifests.

(See Plugins In Modules for info on how to put custom types and facts into modules in the plugins/ subdir)

Example

As an example, consider a autofs module that installs a fixed auto.homes map and generates the auto.master from a template. Its init.pp could look something like:

```
class autofs {
  package { autofs: ensure => latest }
  service { autofs: ensure => running }
  file { "/etc/auto.homes":
    source => "puppet://$servername/modules/autofs/auto.homes"
  }
  file { "/etc/auto.master":
    content => template("autofs/auto.master.erb")
  }
}
```

and have these files in the file system:

```
MODULE_PATH/
  autofs/
    manifests/
      init.pp
    files/
      auto.homes
    templates/
      auto.master.erb
```

Notice that the file source path includes a modules/ component. In Puppet version 0.25 and later, you must include this component in source paths in order to serve files from modules. Puppet 0.25 will still accept source paths without it, but it will warn you with a deprecation notice about "Files found in modules without specifying 'modules' in file path". In versions 0.24 and earlier, source paths should *not* include the modules/ component.

Note also that you can still access files in modules when using puppet instead of puppetd; just leave off the server name and puppetd will fill in the server for you (using its configuration server as its file server) and puppet will use its module path:

```
file { "/etc/auto.homes":
    source => "puppet:///modules/autofs/auto.homes"
}
```

Module Lookup

Since modules contain different subdirectories for different types of files, a little behind-the-scenes magic makes sure that the right file is accessed in the right context. All module searches are done within the modulepath, a colon-separated list of directories. In most cases, searching files in modules amounts to inserting one of manifest, files, or templates after the first component into a path, i.e. paths can be thought of as down-cased_module_name/part_path where part_path is a path relative to one of the subdirectories of the module module_name.

For file references on the fileserver, a similar lookup is used so that a reference to puppet://$servername/modules/autofs/ resolves to the file autofs/files/auto.homes in the module's path. (Note that this behavior will break if you have declared an explicit [autofs] mount in your fileserver.conf, so take care to avoid name collisions when assigning custom fileserver mount points outside of modules.)

You can apply some access controls to files in your modules by creating a [modules] file mount, which should be specified without a path statement, in the fileserver.conf configuration file:

```
[modules]
allow *.domain.com
deny *.wireless.domain.com
```

Unfortunately, you cannot apply more granular access controls, for example at the per module level as yet.

To make a module usable with both the command line client and a puppetmaster, you can use a URL of the form puppet:///path, i.e. a URL without an explicit server name. Such URL's are treated slightly differently by puppet and puppetd: puppet searches for a serverless URL in the local filesystem, and puppetd retrieves such files from the fileserver on the puppetmaster. This makes it possible to use the same module as part of a site manifest on a puppetmaster and in a standalone puppet script by running puppet --modulepath {path} script.pp, without any changes to the module.

Finally, template files are searched in a manner similar to manifests and files: a mention of template("autofs/auto.master will make the puppetmaster first look for a file in $templatedir/autofs/auto.master.erb and then autofs/templates/auto.master.erb on the module path. This allows more-generic files to be provided in the templatedir and more-specific files under the module path (see the discussion under Feature 1012 for the history here).

Module Autoloading

Since version 0.23.1, Puppet will attempt to autoload classes and definitions from modules, so you no longer have to explicitly import them; you can just include the class or start using the definition.

The rules Puppet uses to find the appropriate manifest when a module class or definition is declared are pretty easy to understand, and break down like this:

```
include foo  # {modulepath}/foo/manifests/init.pp

class foo { … }

include foo::bar  # {modulepath}/foo/manifests/bar.pp

class foo::bar { … }

foo::params { "example": value => 'meow' }  # {modulepath}/foo/manifests/params.pp

define foo::params ($value) { … }

class { "foo::bar::awesome": }  # {modulepath}/foo/manifests/bar/awesome.pp

class foo::bar::awesome { … }
```

In short, lookup paths within a module's manifest directory are derived by splitting class and definition names on :: separators, then interpreting the first element as the name of the module, the final element as the filename (with a .pp extension appended), and any intermediate elements as subdirectories of the module's manifests directory:

```
{module name}::{subdirectory}::{...}::{filename (sans extension)}
```

The one special case is that a one-word class or definition name which matches the name of the module will always be found in manifests/init.pp.[1]

Since lookup of classes and definitions is based on filename, take care to always rename both at the same time.

Generated Module Documentation

If you decide to make your modules available to others (and please do!), then please also make sure you document your module so others can understand and use them. Most importantly, make sure the dependencies on other defines and classes not in your module are clear.

From Puppet version 0.24.7 you can generate automated documentation from resources, classes and modules using the puppetdoc tool. You can find more detail at the Puppet Manifest Documentation page.

See Also

Distributing custom facts and types via modules: Plugins In Modules

Writing module tests: Module Smoke Testing

1. Puppet actually always loads the init.pp manifest, so sometimes you can cheat and just write all your module's classes in there. This makes it harder for people to find where your class or define lives, though, so we don't recommend it.

Chapter 19

Using Parameterized Classes

Use parameterized classes to write more effective, versatile, and encapsulated code.

Why, and Some History

Well-written and reusable classes often have to change their behavior based on where and how they're declared. However, due to the organic way the Puppet language grew, there was a long period where it didn't have a specific means to do this.

Most Puppet coders solved this by using dynamic variable lookup to pass parameters into classes. By making the class's effects pivot on a handful of variables not defined in the class, you could later set those variables at node scope or in another class, then declare the class and assign its parent scope; at that point, the class would go looking for the information it needed and react accordingly.

This approach did the job and solved some really important problems, but it had major drawbacks:

- **It basically exploded all variables into the global namespace.** Since classes had to look outside their own scope for parameters, parameters were effectively global. That meant you had to anticipate what every other module author was going to name their variables and try to guess something safe.
- **Understanding how to declare a class was not exactly straightforward.** There was no built-in way to tell what parameters a class needed to have set, so you were on your own for documenting it and following the rules to the letter. Optional parameters in particular could bite you at exactly the wrong time.
- **It was just plain confusing.** The rules for how a parent scope is assigned can fit on an index card, but they can interact in some extraordinarily hairy ways. (ibid.)

So to shorten a long story, Puppet 2.6 introduced a better and more direct way to pass parameters into a class.

Philosophy

A class that depends on dynamic scope for its parameters has to do its own research. Instead, you should supply it with a full dossier when you declare it. Start thinking in terms of passing information to the class, instead of in terms of setting variables and getting scope to act right.

Using Parameterized Classes

Writing a Parameterized Class

Parameterized classes are declared just like classical classes, but with a list of parameters (in parentheses) between the class name and the opening bracket:

```
class webserver( $vhost_dir, $packages ) {
  ...
}
```

The parameters you name can be used as normal local variables throughout the class definition. In fact, the first step in converting a class to use parameters is to just locate all the variables you're expecting to find in an outer scope and call them out as parameters — you won't have to change how they're used inside the class at all.

```
class webserver( $vhost_dir, $packages ) {
  packages { $packages: ensure => present }

  file { 'vhost_dir'
    path   => $vhost_dir,
    ensure => directory,
    mode   => '0750',
    owner  => 'www-data',
    group  => 'root',
  }
}
```

You can also give default values for any parameter in the list:

```
class webserver( $vhost_dir = '/etc/httpd/conf.d', $packages = 'httpd' ) {
  ...
}
```

Any parameter with a default value can be safely omitted when declaring the class.

Declaring a Parameterized Class

This can be easy to forget when using the shorthand include function, but class instances are just resources. Since include wasn't designed for use with parameterized classes, you have to declare them like a normal resource: type, name, and attributes, in their normal order. The parameters you named when defining the class become the attributes you use when declaring it:

```
class {'webserver':
  packages  => 'apache2',
  vhost_dir => '/etc/apache2/sites-enabled',
}
```

Or, if declaring with all default values:

```
class {'webserver': }
```

As of Puppet 2.6.5, parameterized classes can be declared by external node classifiers; see the ENC documentation for details.

Site Design and Composition With Parameterized Classes

Once your classes are converted to use parameters, there's some work remaining to make sure your classes can work well together.

A common pattern with standard classes is to include any other classes that the class requires. Since include ensures a class is declared without redeclaring it, this has been a convenient way to satisfy dependencies. This won't work well with parameterized classes, though, for the reasons we've mentioned above.

Instead, you should explicitly state your class's dependencies inside its definition using the relationship chaining syntax:

```
class webserver( $vhost_dir, $packages ) {
  ...
  # Make sure our ports are configured correctly:
  Class['iptables::webserver'] -> Class['webserver']
}
```

Instead of implicitly declaring the required class, this will make sure that compilation throws an error if it's absent. From one perspective, this is less convenient; from another, it's less magical and more knowable. For those who prefer implicit declaration, we're working on a safe way to implicitly declare parameterized classes, but the design work isn't finished at the time of this writing.

Once you've stated your class's dependencies, you'll need to declare the required classes when composing your node or wrapper class:

```
class tacoma_webguide_application_server {
  class {'webserver':
    packages  => 'apache2',
    vhost_dir => '/etc/apache2/sites-enabled',
  }
  class {'iptables::webserver':}
}
```

The general rule of thumb here is that you should only be declaring other classes in your outermost node or class definitions.

Further Reading

For more information on modern Puppet class and module design, see the Puppet Labs style guide.

Appendix: Smart Parameter Defaults

This design pattern can make for significantly cleaner code while enabling some really sophisticated behavior around default values.

```
# /etc/modules/webserver/manifests/params.pp

class webserver::params {
 $packages = $operatingsystem ? {
   /(?i-mx:ubuntu|debian)/        => 'apache2',
   /(?i-mx:centos|fedora|redhat)/ => 'httpd',
 }
 $vhost_dir = $operatingsystem ? {
   /(?i-mx:ubuntu|debian)/        => '/etc/apache2/sites-enabled',
   /(?i-mx:centos|fedora|redhat)/ => '/etc/httpd/conf.d',
 }
}

# /etc/modules/webserver/manifests/init.pp

class webserver(
 $packages  = $webserver::params::packages,
 $vhost_dir = $webserver::params::vhost_dir
) inherits $webserver::params {

 packages { $packages: ensure => present }

 file { 'vhost_dir'
   path   => $vhost_dir,
   ensure => directory,
   mode   => '0750',
   owner  => 'www-data',
   group  => 'root',
 }
}
```

To summarize what's happening here: When a class inherits from another class, it implicitly declares the base class. Since the base class's local scope already exists before the new class's parameters get declared, those parameters can be set based on information in the base class.

This is functionally equivalent to doing the following:

```
# /etc/modules/webserver/manifests/init.pp

class webserver( $packages = 'UNSET', $vhost_dir = 'UNSET' ) {

 if $packages == 'UNSET' {
   $real_packages = $operatingsystem ? {
     /(?i-mx:ubuntu|debian)/        => 'apache2',
     /(?i-mx:centos|fedora|redhat)/ => 'httpd',
   }
 }
 else {
    $real_packages = $packages
 }

 if $vhost_dir == 'UNSET' {
   $real_vhost_dir = $operatingsystem ? {
     /(?i-mx:ubuntu|debian)/        => '/etc/apache2/sites-enabled',
     /(?i-mx:centos|fedora|redhat)/ => '/etc/httpd/conf.d',
   }
 }
 else {
    $real_vhost_dir = $vhost_dir
}

 packages { $real_packages: ensure => present }

 file { 'vhost_dir'
   path    => $real_vhost_dir,
   ensure => directory,
   mode    => '0750',
   owner  => 'www-data',
   group  => 'root',
 }
}
```

… but it's a significant readability win, especially if the amount of logic or the number of parameters gets any higher than what's shown in the example.

Chapter 20

Module Smoke Testing

Learn to write and run tests for each manifest in your Puppet module.

Doing some basic "Has it exploded?" testing on your Puppet modules is extremely easy, has obvious benefits during development, and can serve as a condensed form of documentation.

Testing in Brief

The baseline for module testing used by Puppet Labs is that each manifest should have a corresponding test manifest that declares that class or defined type.

Tests are then run by using `puppet apply --noop` (to check for compilation errors and view a log of events) or by fully applying the test in a virtual environment (to compare the resulting system state to the desired state).

Writing Tests

A well-formed Puppet module implements each of its classes or defined types in separate files in its `manifests` directory. Thus, ensuring each class or type has a test will result in the `tests` directory being a complete mirror image of the `manifests` directory.

A test for a class is just a manifest that declares the class. Often, this is going to be as simple as `include apache::ssl`. For parameterized classes, the test must declare the class with all of its required attributes set:

```
class {'ntp':
  servers => ['0.pool.ntp.org', '1.pool.ntp.org'],
}
```

Tests for defined resource types may increase test coverage by declaring multiple instances of the type, with varying values for their attributes:

```
dotfiles::user {'root':
  overwrite => false,
}
dotfiles::user {'nick':
  overwrite => append,
}
dotfiles::user {'guest':
  overwrite => true,
}
```

If a class (or type) depends on any other classes, the test will have to declare those as well:

```
# git/manifests/gitosis.pp
class git::gitosis {
  package {'gitosis':
    ensure => present,
  }
  Class['::git'] -> Class['git::gitosis']
}

# git/tests/gitosis.pp
class{'git':}
class{'git::gitosis':}
```

Running Tests

Run tests by applying the test manifests with puppet apply.

For basic smoke testing, you can apply the manifest with `--noop`. This will ensure that a catalog can be properly compiled from your code, and it'll show a log of the RAL events that would have been performed; depending on how simple the class is, these are often enough to ensure that it's doing what you expect.

For more advanced coverage, you can apply the manifest to a live system, preferably a VM. You can expand your coverage further by maintaining a stable of snapshotted environments in various states, to ensure that your classes do what's expected in all the situations where they're likely to be applied.

Automating all this is going to depend on your preferred tools and processes, and is thus left as an exercise for the reader.

Reading Tests

Since module tests declare their classes with all required attributes and with all prerequisites declared, they can serve as a form of drive-by documentation: if you're in a hurry, you can often figure out how to use a module (or just refresh your memory) by skimming through the tests directory.

This doesn't get anyone off the hook for writing real documentation, but it's a good reason to write tests even if your module is already working as expected.

Exploring Further

This form of testing is extremely basic, and still requires a human reader to determine whether the right RAL events are being generated or the right system configuration is being enforced. For more advanced testing, you may want to investigate cucumber-puppet or cucumber-nagios.

Chapter 21

Scope and Puppet as of 2.7

Puppet 2.7 issues deprecation warnings for dynamic variable and resource defaults lookup. Find out why, and learn how to adapt your Puppet code for the future!

What's Changing?

Dynamic scope will be removed from the Puppet language in a future version. **This will be a major and backwards-incompatible change.** Currently, if an unqualified variable isn't defined in the local scope, Puppet looks it up along a chain of parent scopes, eventually ending at top scope; resource defaults (`File{ owner => root , }`, e.g.) travel in much the same way. In the future, Puppet will only examine the local scope and top scope when resolving an unqualified variable or a resource default; intervening scopes will be ignored. **In effect, all variables will be either strictly local or strictly global.** The one exception will be derived classes, which will continue to consult the scope of the base class they inherit from.

To ease the transition, Puppet 2.7 issues deprecation warnings whenever dynamic variable lookup occurs. You should strongly consider refactoring your code to eliminate these warnings.

Why?

Dynamic scope is confusing and dangerous, and often causes unexpected behavior. There are already better methods for accomplishing everything dynamic scope currently does, but even if you're being good, it can step in to "help" at inopportune moments. Dynamic scope interacts really badly with class inheritance, and it makes the boundaries between classes a lot more porous than good programming practice demands.

Thus, it's time to bid it a fond farewell.

Making the Switch

So you've installed Puppet 2.7 and are ready to start going after those deprecation warnings. What do you do?

Qualify Your Variables!

Whenever you need to refer to a variable in another class, give the variable an explicit namespace: instead of simply referring to `$packagelist`, use `$git::core::packagelist`. This is a win in readability — any casual observer can tell exactly where the variable is being set, without having to model your code in their head — and it saves you from accidentally getting the value of some completely unrelated `$packagelist` variable.

If you're referring explicitly to a top-scope variable, use the empty namespace (e.g. `$::packagelist`) for extra clarity.

Declare Resource Defaults Per-File!

If you're using dynamic scope to share resource defaults, there's no way around it: you'll have to repeat yourself in each file that the defaults apply to.

But this is not a bad thing! Resource defaults are really just code compression, and were designed to make a single file of Puppet code more concise. By making sure your defaults are always on the same page as the resources they apply to, you'll make your code vastly more legible and predictable.

If you need to apply resource defaults more broadly, you can still set them at top scope in your primary site manifest. If you need the resource defaults in a class to change depending on where the class is being declared, you need parameterized classes.

All told, it's more likely that defaults have been traveling through scopes without your knowledge, and the eventual elimination of dynamic scope will just make them act like you thought they were acting.

Use Parameterized Classes!

If you need a class to dynamically change its behavior depending on where and how you declare it, it should be rewritten as a parameterized class; see our guide to using parameterized classes for more details.

Appendix: How Scope Works in Puppet 2.7.x

(Note that nodes defined in the Puppet DSL function identically to classes.)

- Classes, nodes, and instances of defined types introduce new scopes.
- When you declare a variable in a scope, it is local to that scope.
- Every scope has one and only one "parent scope."
  - If it's a class that inherits from a base class, its parent scope is the base class.
  - Otherwise, its parent scope is the FIRST scope where that class was declared. (If you are declaring classes in multiple places with include, this can be unpredictable. Furthermore, declaring a derived class will implicitly declare the base class in that same scope.)
- If you try to resolve a variable that doesn't exist in the current local scope, lookup proceeds through the chain of parent scopes — its parent, the parent's parent, and so on, stopping at the first place it finds that variable.

These rules seem simple enough, so an example is in order:

```
# manifests/site.pp
$nodetype = "base"

node "base" {
    include postfix
    ...snip...

}

node "www01", "www02", ... , "www10" inherits "base" {
     $nodetype = "wwwnode"
     include postfix::custom

}

# modules/postfix/manifests/init.pp
# (Template stored in modules/postfix/templates/main.cf.erb)
class postfix {
     package {"postfix": ensure => installed}
     file {"/etc/postfix/main.cf":
          content => template("postfix/main.cf.erb")
     }

}
```

```
# modules/postfix/manifests/custom.pp
class postfix::custom inherits postfix {
     File ["/etc/postfix/main.cf"] {
          content => undef,
          source => [ "puppet:///files/$hostname/main.cf",
                      "puppet:///files/$nodetype/main.cf" ]
     }

}
```

When nodes www01 through www10 connect to the puppet master, $nodetype will always be set to "base" and main.cf will always be served from files/base/. This is because postfix::custom's chain of parent scopes is postfix::custom < postfix < base < top−scope; the combination of inheritance and dynamic scope causes lookup of the $nodetype variable to bypass node 01−10 entirely.

Thanks to Ben Beuchler for contributing this example.

Chapter 22

The Puppet File Server

This guide covers the use of Puppet's file serving capability.

The puppet master service includes a file server for transferring static files. If a file resource declaration contains a puppet: URI in its source attribute, nodes will retrieve that file from the master's file server:

```
# copy a remote file to /etc/sudoers
file { "/etc/sudoers":
    mode => 440,
    owner => root,
    group => root,
    source => "puppet:///modules/module_name/sudoers"
}
```

All puppet file server URIs are structured as follows:

```
puppet://{server hostname (optional)}/{mount point}/{remainder of path}
```

If a server hostname is omitted (i.e. puppet:///{mount point}/{path}; note the triple-slash), the URI will resolve to whichever server the evaluating node considers to be its master. As this makes manifest code more portable and reusable, hostnames should be omitted whenever possible.

The remainder of the puppet: URI maps to the server's filesystem in one of two ways, depending on whether the files are provided by a module or exposed through a custom mount point.

Serving Module Files

As the vast majority of file serving should be done through modules, the Puppet file server provides a special and semi-magical mount point called modules, which is available by default. If a URI's mount point is modules, Puppet will:

- Interpret the next segment of the path as the name of a module…[1]
- … locate that module in the server's modulepath (as described here under "Module Lookup")…
- … and resolve the remainder of the path starting in that module's files/ directory.

That is to say, if a module named test_module is installed in the central server's /etc/puppet/modules directory, the following puppet: URI…

```
puppet:///modules/test_module/testfile.txt
```

…will resolve to the following absolute path:

```
/etc/puppet/modules/test_module/files/testfile.txt
```

If test_module were installed in /usr/share/puppet/modules, the same URI would instead resolve to:

```
/usr/share/puppet/modules/test_module/files/testfile.txt
```

Although no additional configuration is required to use the modules mount point, some access controls can be specified in the file server configuration by adding a [modules] configuration block; see Security.

Serving Files From Custom Mount Points

Puppet can also serve files from arbitrary mount points specified in the server's file server configuration (see below). When serving files from a custom mount point, Puppet does not perform the additional URI abstraction used in the modules mount, and will resolve the path following the mount name as a simple directory structure.

File Server Configuration

The default location for the file server's configuration data is /etc/puppet/fileserver.conf; this can be changed by passing the --fsconfig flag to puppet master.

The format of the fileserver.conf file is almost exactly like that of rsync, and roughly resembles an INI file:

```
[mount_point]
    path /path/to/files
    allow *.domain.com
    deny *.wireless.domain.com
```

The following options can currently be specified for a given mount point:

- The path to the mount's location on the disk
- Any number of allow directives
- Any number of deny directives

path is the only required option, but since the default security configuration is to deny all access, a mount point with no allow directives would not be available to any nodes.

The path can contain any or all of %h, %H, and %d, which are dynamically replaced by the client's hostname, its fully qualified domain name and its domain name, respectively. All are taken from the client's SSL certificate (so be careful if you've got hostname/certname mismatches). This is useful in creating modules where files for each client are kept completely separately, e.g. for private ssh host keys. For example, with the configuration

```
[private]
    path /data/private/%h
    allow *
```

the request for file /private/file.txt from client client1.example.com will look for a file /data/private/client1/file.txt, while the same request from client2.example.com will try to retrieve the file /data/private/client2/file.txt on the fileserver.

Currently paths cannot contain trailing slashes or an error will result. Also take care that in puppet.conf you are not specifying directory locations that have trailing slashes.

Security

Securing the Puppet file server consists of allowing and denying access (at varying levels of specificity) per mount point. Groups of nodes can be identified for permission or denial in three ways: by IP address, by name, or by a single global wildcard (*). Custom mount points default to denying all access.

In addition to custom mount points, there are two special mount points which can be managed with fileserver.conf: modules and plugins. Neither of these mount points should have a path option specified. The behavior of the modules mount point is described above under Serving Files From Custom Mount Points. The plugins mount is not a true mount point, but is rather a hook to allow fileserver.conf to specify which nodes are permitted to sync plugins from the Puppet Master. Both of these mount points exist by default, and both default to allowing all access; if **any** allow or deny directives are set for one of these special mounts, its security settings will behave like those of a normal mount (i.e., it will default to denying all access). Note that these are the only mount points for which deny * is not redundant.

If nodes are not connecting to the Puppet file server directly, e.g. using a reverse proxy and Mongrel (see Using Mongrel), then the file server will see all the connections as coming from the proxy server's IP address rather than that of the Puppet Agent node. In this case, it is best to restrict access based on hostname. Additionally, the machine(s) acting as reverse proxy (usually 127.0.0.0/8) will need to be allowed to access the applicable mount points.

Priority

More specific deny and allow statements take precedence over less specific statements; that is, an allow statement for node.domain.com would let it connect despite a deny statement for *.domain.com. At a given level of specificity, deny statements take precedence over allow statements.

Unpredictable behavior can result from mixing IP address directives with hostname and domain name directives, so try to avoid doing that. (Currently, if node.domain.com's IP address is 192.168.1.80 and fileserver.conf contains allow 192.168.1.80 and deny node.domain.com, the IP-based allow directive will actually take precedence. This behavior may be changed in the future and should not be relied upon.)

Host Names

Host names can be specified using either a complete hostname, or specifying an entire domain using the * wildcard:

```
[export]
    path /export
    allow host.domain1.com
    allow *.domain2.com
    deny badhost.domain2.com
```

IP Addresses

IP address can be specified similarly to host names, using either complete IP addresses or wildcarded addresses. You can also use CIDR-style notation:

```
[export]
    path /export
    allow 127.0.0.1
    allow 192.168.0.*
    allow 192.168.1.0/24
```

Global allow

Specifying a single wildcard will let any node access a mount point:

```
[export]
    path /export
    allow *
```

Note that the default behavior for custom mount points is equivalent to deny *.

1. Older versions of Puppet generated individual mount points for each installed module; to reduce namespace conflicts, these were changed to subdirectories of the catch-all modules mount point in version 0.25.0.

Chapter 23

Style Guide

Style Guide Metadata

Version 1.0.2

Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, ”MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

Puppet Version

This style guide is largely specific to Puppet versions 2.6.x; some of its recommendations are based on some language features that became available in version 2.6.0 and later.

Why a Style Guide?

Puppet Labs develops modules for customers and the community, and these modules should represent the best known practice for module design and style. Since these modules are developed by many people across the organisation, a central reference was needed to ensure a consistent pattern, design, and style.

General Philosophies

No style manual can cover every possible circumstance. When a judgement call becomes necessary, keep in mind the following general ideas:

1. **Readability matters.** If you have to choose between two equally effective alternatives, pick the more readable one. This is, of course, subjective, but if you can read your own code three months from now, that’s a great start.
2. **Inheritance should be avoided.** In general, inheritance leads to code that is harder to read. Most use cases for inheritance can be replaced by exposing class parameters that can be used to configure resource attributes. See the Class Inheritance section for more details.
3. **Modules must work with an ENC without requiring one.** An internal survey yielded near consensus that an ENC should not be required. At the same time, every module we write should work well with an ENC.

4. **Classes should generally not declare other classes.** Declare classes as close to node scope as possible. Classes which require other classes should not directly declare them and should instead allow the system to fail if they are not declared by some other means. (Although the include function allows multiple declarations of classes, it can result in non-deterministic scoping issues due to the way parent scopes are assigned. We might revisit this philosophy in the future if class multi-declarations can be made deterministic, but for now, be conservative with declarations.)

Module Metadata

Every module must have Metadata defined in the Modulefile data file and outputted as the metadata.json file. The following Metadata should be provided for all modules:

```
name 'myuser-mymodule'
version '0.0.1'
author 'Author of the module - for shared modules this is Puppet Labs'
summary 'One line description of the module'
description 'Longer description of the module including an example'
license 'The license the module is release under - generally GPLv2 or Apache'
project_page 'The URL where the module source is located'
dependency 'otheruser-othermodule', '1.2.3'
```

A more complete guide to the Modulefile format can be found in the puppet-module-tool README.

Style Versioning

This style guide will be versioned, which will allow modules to comply with a specific version of the style guide.

A future version of the puppet-module tool may permit the relevant style guide version to be embedded as metadata in the Modulefile, and the metadata in turn may be used for automated linting.

Spacing, Indentation, & Whitespace

Module manifests complying with this style guide:

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

Comments

Although the Puppet language allows multiple comment types, we prefer hash/octothorpe comments (`# This is a comment`) because they're generally the most visible to text editors and other code lexers.

1. Should use `# ...` for comments
2. Should not use `// ...` or `/* ... */` for comments

Quoting

All strings that do not contain variables should be enclosed in single quotes. Double quotes should be used when variable interpolation is required. Quoting is optional when the string is an alphanumeric bare word and is not a resource title.

All variables should be enclosed in braces when interpolated in a string. For example:

Good:

```
"/etc/${file}.conf"
"${operatingsystem} is not supported by ${module_name}"
```

Bad:

```
"/etc/$file.conf"
"$operatingsystem is not supported by $module_name"
```

Variables standing by themselves should not be quoted. For example:

Good:

```
mode => $my_mode
```

Bad:

```
mode => "$my_mode"
mode => "${my_mode}"
```

Resources

Resource Names

All resource titles should be quoted. (Puppet supports unquoted resource titles if they do not contain spaces or hyphens, but you should avoid them in the interest of consistent look-and-feel.)

Good:

```
package { 'openssh': ensure => present }
```

Bad:

```
package { openssh: ensure => present }
```

Arrow Alignment

All of the fat comma arrows (=>) in a resource's attribute/value list should be aligned. The arrows should be placed one space ahead of the longest attribute name.

Good:

```
exec { 'blah':
  path => '/usr/bin',
  cwd  => '/tmp',
}

exec { 'test':
  subscribe   => File['/etc/test'],
  refreshonly => true,
}
```

Bad:

```
exec { 'blah':
 path => '/usr/bin',
 cwd => '/tmp',
}

exec { 'test':
  subscribe => File['/etc/test'],
  refreshonly => true,
}
```

Attribute Ordering

If a resource declaration includes an ensure attribute, it should be the first attribute specified.

Good:

```
file { '/tmp/readme.txt':
  ensure => file,
  owner  => '0',
  group  => '0',
  mode   => '0644',
}
```

(This recommendation is solely in the interest of readability, as Puppet ignores attribute order when syncing resources.)

Compression

Within a given manifest, resources should be grouped by logical relationship to each other, rather than by resource type. Use of the semicolon syntax to declare multiple resources within a set of curly braces is not recommended, except in the rare cases where it would improve readability.

Good:

```
file { '/tmp/a':
  content => 'a',
}

exec { 'change contents of a':
  command => 'sed -i.bak s/a/A/g /tmp/a',
}

file { '/tmp/b':
  content => 'b',
}

exec { 'change contents of b':
  command => 'sed -i.bak s/b/B/g /tmp/b',
}
```

Bad:

```
file {
  "/tmp/a":
    content => "a";
  "/tmp/b":
    content => "b";
}

exec {
  "change contents of a":
    command => "sed -i.bak s/b/B/g /tmp/a";
  "change contents of b":
    command => "sed -i.bak s/b/B/g /tmp/b";
}
```

Symbolic Links

In the interest of clarity, symbolic links should be declared by using an ensure value of ensure => link and explicitly specifying a value for the target attribute. Using a path to the target as the ensure value is not recommended.

Good:

```
file { '/var/log/syslog':
  ensure => link,
  target => '/var/log/messages',
}
```

Bad:

```
file { '/var/log/syslog':
  ensure => '/var/log/messages',
}
```

File Modes

File modes should be represented as 4 digits rather than 3, to explicitly show that they are octal values.

In addition, file modes should be specified as single-quoted strings instead of bare word numbers.

Good:

```
file { '/var/log/syslog':
  ensure => present,
  mode   => '0644',
}
```

Bad:

```
file { '/var/log/syslog':
  ensure => present,
  mode   => 644,
}
```

Resource Defaults

Resource defaults should be used in a very controlled manner, and should only be declared at the edges of your manifest ecosystem. Specifically, they may be declared:

- At top scope in site.pp
- In a class which is guaranteed to never declare another class and never be inherited by another class.

This is due to the way resource defaults propagate through dynamic scope, which can have unpredictable effects far away from where the default was declared.

Good:

```
# /etc/puppetlabs/puppet/manifests/site.pp:
File {
  mode   => '0644',
  owner  => 'root',
  group  => 'root',
}
```

Bad:

```
# /etc/puppetlabs/puppet/modules/ssh/manifests/init.pp
File {
  mode   => '0600',
  owner  => 'nobody',
  group  => 'nogroup',
}

class {'ssh::client':
  ensure => present,
}
```

Conditionals

Keep Resource Declarations Simple

You should not intermingle conditionals with resource declarations. When using conditionals for data assignment, you should separate conditional code from the resource declarations.

Good:

```
$file_mode = $operatingsystem ? {
  debian => '0007',
  redhat => '0776',
  fedora => '0007',
}

file { '/tmp/readme.txt':
   content => "Hello World\n",
   mode    => $file_mode,
}
```

Bad:

```
file { '/tmp/readme.txt':
  mode => $operatingsystem ? {
    debian => '0777',
    redhat => '0776',
    fedora => '0007',
  }
}
```

Defaults for Case Statements and Selectors

Case statements should have default cases. Additionally, the default case should fail the catalog compilation when the resulting behavior cannot be predicted on the majority of platforms the module will be used on. If you want the default case to be "do nothing," include it as an explicit `default: {}` for clarity's sake.

For selectors, default selections should only be omitted if you explicitly want catalog compilation to fail when no value matches.

The following example follows the recommended style:

```
case $operatingsystem {
  centos: {
    $version = '1.2.3'
  }
  solaris: {
    $version = '3.2.1'
  }
  default: {
    fail("Module $module_name is not supported on $operatingsystem")
  }
}
```

Classes

Separate Files

All classes and resource type definitions must be in separate files in the `manifests` directory of their module. For example:

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

# init.pp
  class apache { }
# ssl.pp
  class apache::ssl { }
# virtual_host.pp
  define apache::virtual_host () { }
```

This is functionally identical to declaring all classes and defines in init.pp, but highlights the structure of the module and makes everything more legible.

Internal Organization of a Class

Classes should be organised with a consistent structure and style. In the below list there is an implicit statement of "should be at this relative location" for each of these items. The word "may" should be interpreted as "If there are any X's they should be here".

1. Should define the class and parameters
2. Should validate any class parameters and fail catalog compilation if any parameters are invalid
3. Should default any validated parameters to the most general case
4. May declare local variables
5. May declare relationships to other classes Class['apache'] -> Class['local_yum']
6. May override resources
7. May declare resource defaults
8. May declare resources; resources of defined and custom types should go before those of core types
9. May declare resource relationships inside of conditionals

The following example follows the recommended style:

```
class myservice($ensure='running') {

  if $ensure in [ running, stopped ] {
    $ensure_real = $ensure
  } else {
    fail('ensure parameter must be running or stopped')
  }

  case $operatingsystem {
    centos: {
      $package_list = 'openssh-server'
    }
    solaris: {
      $pacakge_list = [ SUNWsshr, SUNWsshu ]
    }
    default: {
      fail("module $module_name does not support $operatingsystem")
    }
  }

  $variable = 'something'

  Package { ensure => present, }

  File { owner => '0', group => '0', mode => '0644' }

  package { $package_list: }

  file { "/tmp/${variable}":
    ensure => present,
  }

  service { 'myservice':
    ensure    => $ensure_real,
    hasstatus => true,
  }
}
```

Relationship Declarations

Relationship declarations with the chaining syntax should only be used in the "left to right" direction.

Good:

```
Package['httpd'] -> Service['httpd']
```

Bad:

```
Service['httpd'] <- Package['httpd']
```

When possible, you should prefer metaparameters to relationship declarations. One example where metaparameters aren't desirable is when subclassing would be necessary to override behavior; in this situation, relationship declarations inside of conditionals should be used.

Classes and Defined Resource Types Within Classes

Classes and defined resource types must not be defined within other classes.

Bad:

```
class apache {
  class ssl { ... }
}
```

Also bad:

```
class apache {
  define config() { ... }
}
```

Class Inheritance

Inheritance may be used within a module, but must not be used across module namespaces. Cross-module dependencies should be satisfied in a more portable way that doesn't violate the concept of modularity, such as with include statements or relationship declarations.

Good:

```
class ssh { ... }

class ssh::client inherits ssh { ... }

class ssh::server inherits ssh { ... }

class ssh::server::solaris inherits ssh::server { ... }
```

Bad:

```
class ssh inherits server { ... }

class ssh:client inherits workstation { ... }

class wordpress inherits apache { ... }
```

Inheritance in general should be avoided when alternatives are viable. For example, instead of using inheritance to override relationships in an existing class when stopping a service, consider using a single class with an ensure parameter and conditional relationship declarations:

```
class bluetooth($ensure=present, $autoupgrade=false) {
  # Validate class parameter inputs. (Fail early and fail hard)

  if ! ($ensure in [ "present", "absent" ]) {
    fail("bluetooth ensure parameter must be absent or present")
  }

  if ! ($autoupgrade in [ true, false ]) {
```

```
    fail("bluetooth autoupgrade parameter must be true or false")
  }

  # Set local variables based on the desired state

  if $ensure == "present" {
    $service_enable = true
    $service_ensure = running
    if $autoupgrade == true {
      $package_ensure = latest
    } else {
      $package_ensure = present
    }
  } else {
    $service_enable = false
    $service_ensure = stopped
    $package_ensure = absent
  }

  # Declare resources without any relationships in this section

  package { [ "bluez-libs", "bluez-utils"]:
    ensure => $package_ensure,
  }

  service { hidd:
    enable        => $service_enable,
    ensure        => $service_ensure,
    status        => "source /etc/init.d/functions; status hidd",
    hasstatus     => true,
    hasrestart    => true,
  }

  # Finally, declare relations based on desired behavior

  if $ensure == "present" {
    Package["bluez-libs"]  -> Package["bluez-utils"]
    Package["bluez-libs"]  ~> Service[hidd]
    Package["bluez-utils"] ~> Service[hidd]
  } else {
    Service["hidd"]        -> Package["bluez-utils"]
    Package["bluez-utils"] -> Package["bluez-libs"]
  }
}
```

(This example makes several assumptions and is based on an example provided in the Puppet Master training for managing bluetooth.)

In summary:

- Class inheritance is only useful for overriding resource attributes; any other use case is better accomplished with other methods.
- If you just need to override relationship metaparameters, you should use a single class with conditional relationship declarations instead of inheritance.
- In many cases, even other attributes (e.g. ensure and enable) may have their behavior changed with variables and conditional logic instead of inheritance.

Namespacing Variables

When using top-scope variables, including facts, Puppet modules should explicitly specify the empty namespace (i.e., `$::operatingsystem`, not `$operatingsystem`) to prevent accidental scoping issues.

Display Order of Class Parameters

In parameterized class and defined resource type declarations, parameters that are required should be listed before optional parameters (i.e. parameters with defaults).

Good:

```
class ntp (
  $servers,
  $options = "iburst",
  $multicast = false
) {}
```

Bad:

```
class ntp (
  $options = "iburst",
  $servers,
  $multicast = false
) {}
```

Tests

All manifests should have a corresponding test manifest in the module's tests directory.

```
modulepath/apache/manifests/{init,ssl}.pp
modulepath/apache/tests/{init,ssl}.pp
```

The test manifest should provide a clear example of how to declare the class or defined resource type. In addition, the test manifest should also declare any classes required by the corresponding class to ensure puppet apply works in a limited, stand alone manner.

Puppet Doc

Classes and defined resource types should be documented inline using the following conventions:

For classes:

```
# Full description of class here.
#
# == Parameters
#
# Document parameters here
#
# [*servers*]
#   Description of servers class parameter.  e.g. "Specify one or more
#   upstream ntp servers as an array."
#
# == Variables
#
# Here you should define a list of variables that this module would require.
#
# [*$enc_ntp_servers*]
#     Description of this variable.  e.g. "The parameter enc_ntp_servers
# must be set by the External Node Classifier as a comma separated list of
# hostnames."  (Note, global variables should not be used in preference to
# class parameters as of Puppet 2.6.)
#
# == Examples
#
# Put some examples on how to use your class here.
#
#   $example_var = "blah"
#   include example_class
#
# == Authors
#
# Author Name <author@domain.com\>
#
# == Copyright
#
# Copyright 2011 Company Name Inc, unless otherwise noted.
#
class example_class {
```

```
  ...
}
```

For defined resources:

```
# Description of resource here
#
# == Parameters
#
# Document parameters here
#
# [*namevar*]
#   If there is a parameter that defaults to the value of the title string
# when not explicitly set, you must always say so.  This parameter can be
# referred to as a "namevar," since it's functionally equivalent to the
# namevar of a core resource type.
#
# [*basedir*]
#   Description of this variable.  For example, "This parameter sets the
# base directory for this resource type.  It should not contain a trailing
# slash."
#
# == Examples
#
# Provide some examples on how to use this type:
#
#   example_class::example_resource{
#     "namevar":
#       sample_param => "value",
#   }
#
define example_class::example_resource($example_var) {
  ...
}
```

This will allow documentation to be automatically extracted with the puppet doc tool.

Chapter 24

Best Practices

This guide includes some tips to getting the most out of Puppet. It is derived from the best practices section of the Wiki and other sources. It is intended to cover high-level best practices and may not extend into lower level details.

Use Modules When Possible

Puppet modules are something everyone should use. If you have an application you are managing, add a module for it, so that you can keep the manifests, plugins (if any), source files, and templates together.

Keep Your Puppet Content In Version Control

Keep your Puppet manifests in version control. You can pick your favorite systems — popular choices include git and svn.

Naming Conventions

Node names should match the hostnames of the nodes.

When naming classes, a class that disables ssh should be inherited from the ssh class and be named "ssh::disabled"

Style

For recommendations on syntax and formatting, follow the Style Guide

Classes Vs Defined Types

Classes are not to be thought of in the 'object oriented' meaning of a class. This means a machine belongs to a particular class of machine.

For instance, a generic webserver would be a class. You would include that class as part of any node that needed to be built as a generic webserver. That class would drop in whatever packages, etc, it needed to do.

Defined types on the other hand (created with 'define') can have many instances on a machine, and can encapsulate classes and other resources. They can be created using user supplied variables. For instance, to manage iptables, a defined type may wrap each rule in the iptables file, and the iptables configuration could be built out of fragments generated by those defined types.

Usage of classes and defined types, in addition to the built-in managed types, is very helpful towards having a managable Puppet infrastructure.

Work In Progress

This document is a stub. You can help Puppet by submitting contributions to it.

Chapter 25

Using Puppet Templates

Learn how to template out configuration files with Puppet, filling in variables from the client system from facter.

Puppet supports templates and templating via ERB, which is part of the Ruby standard library and is used for many other projects including Ruby on Rails. While it is a Ruby templating system, you do not need to understand much Ruby to use ERB.

Templates allow you to manage the content of template files, for example configuration files that cannot yet be managed directly by a built-in Puppet type. This might include an Apache configuration file, Samba configuration file, etc.

Evaluating templates

Templates are evaluated via a simple function:

```
$value = template("mytemplate.erb")
```

You can specify the full path to your template, or you can put all your templates in Puppet's templatedir, which usually defaults to /var/puppet/templates (you can find out what it is on your system by running puppet --configprint templatedir). Best practices indicates including the template in the templates directory inside your module.

Templates are always evaluated by the parser, not by the client. This means that if you are using puppetmasterd, then the templates only need to be on the server, and you never need to download them to the client. There's no difference that the client sees between using a template and specifying all of the text of the file as a string. This also means that any client-specific variables (facts) are learned first by puppetmasterd during the client start-up phase, then those variables are available for substitution within templates.

Using templates

Here is an example for generating the Apache configuration for Trac sites:

```
define tracsite($cgidir, $tracdir) {
    file { "trac-$name":
        path => "/etc/apache2/trac/$name.conf",
        owner => root,
        group => root,
        mode => 644,
        require => File[apacheconf],
        content => template("tracsite.erb"),
        notify => Service[apache2]
    }

    symlink { "tracsym-$name":
```

```
        path => "$cgidir/$name.cgi",
        ensure => "/usr/share/trac/cgi-bin/trac.cgi"
    }
}
```

And then here's the template:

```
<Location "/cgi-bin/<%= name %>.cgi">
    SetEnv TRAC_ENV "/export/svn/trac/<%= name %>"
</Location>

# You need something like this to authenticate users
<Location "/cgi-bin/<%= name %>.cgi/login">
    AuthType Basic
    AuthName "Trac"
    AuthUserFile /etc/apache2/auth/svn
    Require valid-user
</Location>
```

This puts each Trac configuration into a separate file, and then we just tell Apache to load all of these files:

```
Include /etc/apache2/trac/[^.#]*
```

Combining templates

You can also concatentate several templates together as follows:

```
template('/path/to/template1','/path/to/template2')
```

Iteration

Puppet's templates also support array iteration. If the variable you are accessing is an array, you can iterate over it in a loop. Given Puppet manifest code like this:

```
$values = [val1, val2, otherval]
```

You could have a template like this:

```
<% values.each do |val| -%>
Some stuff with <%= val %>
<% end -%>
```

This would produce:

```
Some stuff with val1
Some stuff with val2
Some stuff with otherval
```

Note that normally, ERB template lines that just have code on them would get translated into blank lines. This is because ERB generates newlines by default. To prevent this, we use the closing tag -%> instead of %>.

As we mentioned, erb is a Ruby system, but you don't need to know Ruby well to use ERB. Internally, Puppet's values get translated to real Ruby values, including true and false, so you can be pretty confident that variables will behave as you might expect.

Conditionals

The ERB templating supports conditionals. The following construct is a quick and easy way to conditionally put content into a file:

```
<% if broadcast != "NONE" %>          broadcast <%= broadcast %> <% end %>
```

Templates and variables

You can also use templates to fill in variables in addition to filling out file contents.

```
myvariable = template('/var/puppet/template/myvar')
```

Undefined variables

If you need to test to see if a variable is defined before using it, the following works:

```
<% if has_variable?("myvar") then %>
myvar has <%= myvar %> value
<% end %>
```

Out of scope variables

You can access out of scope variables explicitly with the lookupvar function:

```
<%= scope.lookupvar('apache::user') %>
```

Access to defined tags and classes

In Puppet version 0.24.6 and later, it is possible from a template to get the list of defined classes, the list of tags in the current scope, and the list of all tags as ruby arrays. For example:

This snippet will print all the tags defined in the current scope:

```
<% tags.each do |tag| -%>
The tag <%= tag %> is part of the current scope
<% end -%>
```

This snippet will print all the defined tags in the catalog:

```
<% all_tags.each do |tag| -%>
The tag <%= tag %> is defined
<% end -%>
```

This snippet will print all the defined classes in the catalog:

```
<% classes.each do |klass| -%>
The class <%= klass %> is defined
<% end -%>
```

Access to variables and Puppet functions with the scope object

Inside templates you have access to a scope object. All of the functions that you can access in the puppet manifests can be accessed via that scope object, although not via the same name.

Variables defined in the current scope are available as entries in the hash returned by the scope object's to_hash method. This snippet will print all of the variable names defined in the current scope:

```
<% scope.to_hash.keys.each do |k| -%>
<%= k %>
<% end -%>
```

Puppet functions can be called by prepending "function_" to the beginning of the function name. For example, including one template inside another:

```
<%= scope.function_template("module/template2.erb") %>
```

Syntax Checking

ERB files are easy to syntax check. For a file mytemplate.erb, run

```
erb -x -T '-' mytemplate.erb | ruby -c
```

Chapter 26

Virtual Resources

Referencing an entity from more than one place.

About Virtual Resources

By default, any resource you describe in a client's Puppet config will get sent to the client and be managed by that client. However, resources can be specified in a way that marks them as virtual, meaning that they will not be sent to the client by default. You mark a resource as virtual by prefixing @ to the resource specification; for instance, the following code defines a virtual user:

```
@user { luke: ensure => present }
```

If you include this code (or something similar) in your configuration then the user will never get sent to your clients without some extra effort.

How This Is Useful

Puppet enforces configuration normalization, meaning that a given resource can only be specified in one part of your configuration. You can't configure user johnny in both the solaris and freebsd classes.

For most cases, this is fine, because most resources are distinctly related to a single Puppet class — they belong in the webserver class, mailserver class, or whatever. Some resources can not be cleanly tied to a specific class, though; multiple otherwise-unrelated classes might need a specific resource. For instance, if you have a user who is both a database administrator and a Unix sysadmin, you want the user installed on all machines that have either database administrators or Unix administrators.

You can't specify the user in the dba class nor in the sysadmin class, because that would not get the user installed for all cases that matter.

In these cases, you can specify the user as a virtual resource, and then mark the user as real in both classes. Thus, the user is still specified in only one part of your configuration, but multiple parts of your configuration verify that the user will be installed on the client.

The important point here is that you can take a virtual resource and mark it non-virtual as many times as you want in a configuration; it's only the specification itself that must be normalized to one specific part of your configuration.

How to Realize Resources

There are two ways to mark a virtual resource so that it gets sent to the agent: You can use a special syntax called a **collection,** or you can use the **realize** function.

Collections provide a simple syntax (sometimes referred to as the "spaceship" operator) for marking virtual objects as real, such that they should be sent to the agent. Collections require the type of resource you are collecting and zero or more attribute comparisons to specifically select resources. For instance, to find our mythical user, we would use:

```
User <| title == luke |>
```

As promised, we've got the user type (capitalized, because we're performing a type-level operation), and we're looking for the user whose title is luke. "Title" is special here — it is the value before the colon when you specify the user. This is somewhat of an inconsistency in Puppet, because this value is often referred to as the name, but many types have a name parameter and they could have both a title and a name.

If no comparisons are specified, all virtual resources of that type will be marked real.

This attribute querying syntax is currently very simple. The only comparisons available are equality and non-equality (using the == and != operators, respectively), and you can join these comparisons using or and and. You can also parenthesize these statements, as you might expect. So, a more complicated collection might look like:

```
User <| (group == dba or group == sysadmin) or title == luke |>
```

Realizing Resources

Puppet provides a simple form of syntactic sugar for marking resource non-virtual by title, the realize function:

```
realize User[luke]
realize(User[johnny], User[billy])
```

The function follows the same syntax as other functions in the language, except that only resource references are valid values.

Virtual Define-Based Resources

Since version 0.23, define-based resources may also be made virtual. For example:

```
define msg($arg) {
  notify { "$name: $arg": }
}
@msg { test1: arg => arg1 }
@msg { test2: arg => arg2 }
```

With the above definitions, neither of the msg resources will be applied to a node unless it realizes them, e.g.:

```
realize( Msg[test1], Msg[test2] )
```

Remember that when referencing an instance of a namespaced defined type, or when specifying such a defined type for the collection syntax, you have to capitalize all segments of the type's name (e.g. Apache::Vhost['wordpress'] or Apache::Vhost <| |>).

Keep in mind that resources inside virtualized define-based resources must have unique names. The following example will fail, complaining that File[foo] is defined twice:

```
define basket($arg) {
        file{'foo':
                ensure  => present,
                content => "$arg",
                }
        }
@basket { 'fruit': arg => 'apple' }
@basket { 'berry': arg => 'watermelon' }

realize( Basket[fruit], Basket[berry] )
```

Here's a working example:

```
define basket($arg) {
    file{"$name":
        ensure  => present,
        content => "$arg",
        }
    }
@basket { 'fruit': arg => 'apple' }
@basket { 'berry': arg => 'watermelon' }

realize( Basket[fruit], Basket[berry] )
```

Note that the working example will result in two File resources, named fruit and berry.

Chapter 27

Exporting and Collecting Resources

Exporting and collecting resources is an extension of Virtual Resources . Puppet provides an experimental superset of virtual resources, using a similar syntax. In addition to these resources being virtual, they're also "exported" to other hosts on your network.

About Exported Resources

While virtual resources can only be collected by the host that specified them, exported resources can be collected by any host. You **must** set the storeconfigs setting to true to enable this functionality (you can see information about stored configuration on the Using Stored Configuration wiki page, and Puppet will automatically create a database for storing configurations (using Ruby on Rails).

```
[puppetmasterd]
storeconfigs = true
```

This allows one host to configure another host; for instance, a host could configure its services using Puppet, and then could export Nagios configurations to monitor those services.

The key syntactical difference between virtual and exported resources is that the special sigils (@ and <| |>) are doubled (@@ and <<| |>>) when referring to an exported resource.

Here is an example with exported resources:

```
class ssh {
    @@sshkey { $hostname: type => dsa, key => $sshdsakey }
    Sshkey <<| |>>
}
```

As promised, we use two @ sigils here, and the angle brackets are doubled in the collection.

The above code would have every host export its SSH public key, and then collect every host's key and install it in the ssh_known_hosts file (which is what the sshkey type does); this would include the host doing the exporting.

It's important to mention here that you will only get exported resources from hosts whose configurations have been compiled. If hostB exports a resource but hostB has never connected to the server, then no host will get that exported resource. The act of compiling a given host's configuration puts the resources into the database, and only resources in the database are available for collection.

Let's look at another example, this time using a File resource:

```
node a {
    @@file { "/tmp/foo": content => "fjskfjs\n", tag => "foofile", }
}
node b {
    File <<| tag == 'foofile' |>>
}
```

This will create /tmp/foo on node b. Note that the tag is not required, it just allows you to control which resources you want to import.

Exported Resources with Nagios

Puppet includes native types for managing Nagios configuration files. These types become very powerful when you export and collect them. For example, you could create a class for something like Apache that adds a service definition on your Nagios host, automatically monitoring the web server:

```
class nagios-target {
   @@nagios_host { $fqdn:
        ensure => present,
        alias => $hostname,
        address => $ipaddress,
        use => "generic-host",
   }
   @@nagios_service { "check_ping_${hostname}":
        check_command => "check_ping!100.0,20%!500.0,60%",
        use => "generic-service",
        host_name => "$fqdn",
        notification_period => "24x7",
        service_description => "${hostname}_check_ping"
   }
}
class nagios-monitor {
    package { [ nagios, nagios-plugins ]: ensure => installed, }
    service { nagios:
        ensure => running,
        enable => true,
        #subscribe => File[$nagios_cfgdir],
        require => Package[nagios],
    }
    # collect resources and populate /etc/nagios/nagios_*.cfg
    Nagios_host <<||>>
    Nagios_service <<||>>
}
```

Exported Resources Override

Beginning in version 0.25, some new syntax has been introduced that allows creation of collections of any resources, not just virtual ones, based on filter conditions, and override of attributes in the created collection. This feature is not constrained to the override in inherited context, as is the case in the usual resource override.

Ordinary resource collections can now be defined by filter conditions, in the same way as collections of virtual or exported resources. For example:

```
file {
    "/tmp/testing": content => "whatever"
}

File<| |> {
    mode => 0600
}
```

The filter condition goes in the middle of the <| |> sigils. In the above example the condition is empty, so all file resources (not just virtual ones) are selected, and all file resources will have their modes overridden to 0600.

In the past this syntax only collected virtual resources. It now collects all matching resources, virtual or no, and allows you to override attributes in any of the collection so defined.

As another example, one can write:

```
file { "/tmp/a": content => "a" }
file { "/tmp/b": content => "b" }

File <| title != "/tmp/b" |> {
    require => File["/tmp/b"]
}
```

This means that every File resource requires /tmp/b, except /tmp/b itself. Moreover, it is now possible to define resource overriding without respecting the override on inheritance rule:

```
class a {
    file {
        "/tmp/testing": content => "whatever"
    }
}

class b {
    include a
    File<| |> {
        mode => 0600
    }
}
include b
```

Chapter 28

Environments

Manage your module releases by dividing your site into environments.

Slice and Dice

Puppet lets you slice your site up into an arbitrary number of "environments" and serve a different set of modules to each one. This is usually used to manage releases of Puppet modules by testing them against scratch nodes before rolling them out completely, but it introduces a lot of other possibilities, like separating a DMZ environment, splitting coding duties among multiple sysadmins, or dividing the site by hardware type.

What an Environment Is

Every agent node has an environment, and the puppet master gets informed about it whenever that node makes a request. (If you don't specify an environment, the agent has the default "production" environment.)

The puppet master can then use that environment several ways:

- If the master's `puppet.conf` file has a [config block] for this agent's environment, those settings will override the master's normal settings when serving that agent.
- If the values of any settings in puppet.conf reference the $environment variable (like modulepath = $confdir/environments/$environment/modules:$confdir/modules, for example), the agent's environment will be interpolated into them.
- Depending on how `auth.conf` is configured, different requests might be allowed or denied.
- The agent's environment will also be accessible in Puppet manifests as the top-scope $environment variable.

In short: modules and manifests can already do different things for different nodes, but environments let the master tweak its own configuration on the fly, and offer a way to completely swap out the set of available modules for certain nodes.

Caveats

Before you start, be aware that environments have some limitations, most of which are known bugs or vagaries of implementation rather than design choices.

- Puppet will only read the `modulepath`, `manifest`, `manifestdir`, and `templatedir` settings from environment config blocks; other settings in any of these blocks will be ignored in favor of settings in the [master] or [main] blocks. (Issue 7497)

- File serving only works well with environments if you're only serving files from modules; if you've set up custom mount points in fileserver.conf, they won't work in your custom environments. (Though hopefully you're only serving files from modules anyway.)
- You can set an agent node's environment from an external node classifier like Puppet Dashboard, but it isn't well-supported: currently, the server-set environment will win during catalog compilation, but the client-set environment will win when downloading files. (Issue 3910) For environments to work reliably, they have to be specified in the agent's configuration.
- Serving custom types and providers from an environment-specific modulepath sometimes fails. (Issue 4409)

Configuring Environments on the Puppet Master

In `puppet.conf`

As mentioned above, puppet.conf lets you use $environment as a variable and create config blocks for environments.

```
# /etc/puppet/puppet.conf
[master]
  modulepath = $confdir/environments/$environment/modules:$confdir/modules
  manifest = $confdir/manifests/unknown_environment.pp
[production]
  manifest = $confdir/manifests/site.pp
[dev]
  manifest = $confdir/manifests/site.pp
```

In the [master] block, this example dynamically sets the modulepath so Puppet will check a per-environment folder for a module before serving it from the main set. Note that this won't complain about missing directories, so you can create the per-environment folders lazily as you need them.

The example also redirects requests for a non-existent environment to a different site manifest, which will log an error and fail compilation; this can keep typos or forgetfulness from silently causing odd configurations.

In `auth.conf`

```
path /
auth any
environment appdev
allow localhost, customapp.puppetlabs.lan
```

If you specify an environment in an `auth.conf` ACL, it will only apply to requests in that environment. This can be useful for developing new applications that integrate with Puppet; the example above will leave normal requests functioning normally, but allow an app server to access everything via the REST API.

In Manifests

The $environment variable should only rarely be necessary, but it's there if you need it.

Configuring Environments for Agent Nodes

To set an environment agent-side, just specify the environment setting in either the [agent] or [main] block of puppet.conf.

```
[agent]
  environment = dev
```

As with any config setting, you can also use a command line option:

```
# puppet agent —environment dev
```

You can also set an environment via your ENC by including an environment: dev (or similar) line in the yaml it returns, but see the caveat above before doing this.

Eventually, server-side environments will work properly, but if you need to work around this today, you can do so by managing puppet.conf on agent nodes with a template. This can take multiple runs to reach the desired configuration for the first time, but it will work.

Compatibility Notes

Environments were introduced in Puppet 0.24.0.

Chapter 29

Reporting

How to learn more about the activity of your nodes.

Reports and Reporting

Puppet clients can be configured to send reports at the end of every configuration run. Because the Transaction interals of Puppet are responsible for creating and sending the reports, these are called transaction reports. Currently, these reports include all of the log messages generated during the configuration run, along with some basic metrics of what happened on that run. In Rowlf, more detailed reporting information will be available, allowing users to see detailed change information regarding what happened on nodes.

Logs

The bulk of the report is every log message generated during the transaction. This is a simple way to send almost all client logs to the Puppet server; you can use the log report to send all of these client logs to syslog on the server.

Metrics

The rest of the report contains some basic metrics describing what happened in the transaction. There are three types of metrics in each report, and each type of metric has one or more values:

- **Time: Keeps track of how long things took.**
  - *Total*: Total time for the configuration run
  - *File*:
  - *Exec*:
  - *User*:
  - *Group*:
  - *Config Retrieval*: How long the configuration took to retrieve
  - *Service*:
  - *Package*:
- **Resources: Keeps track of the following stats:**
  - *Total*: The total number of resources being managed
  - *Skipped*: How many resources were skipped, because of either tagging or scheduling restrictions
  - *Scheduled*: How many resources met any scheduling restrictions
  - *Out of Sync*: How many resources were out of sync
  - *Applied*: How many resources were attempted to be fixed
  - *Failed*: How many resources were not successfully fixed

  - *Restarted*: How many resources were restarted because their dependencies changed
  - *Failed Restarts*: How many resources could not be restarted

- **Changes**: The total number of changes in the transaction.

Setting Up Reporting

By default, the client does not send reports, and the server only is only configured to store reports, which just stores recieved YAML-formatted report in the reportdir.

Clients default to sending reports to the same server they get their configurations from, but you can change that by setting reportserver on the client, so if you have load-balanced Puppet servers you can keep all of your reports consolidated on a single machine.

Sending Reports

In order to turn on reporting on the client-side (puppetd), the report argument must be given to the puppetd executable either by passing the argument to the executable on the command line, like this:

```
$ puppetd —report
```

or by including the configuration parameter in the Puppet configuration file, usually located in /etc/puppet/puppet.conf:

```
#
#  /etc/puppet/puppet.conf
#
[puppetd]
    report = true
```

With this setting enabled, the client will then send the report to the puppetmasterd server at the end of every transaction.

If you are using namespaceauth.conf, you must allow the clients to access the name space:

```
#
# /etc//puppet/namespaceauth.conf
#
[puppetreports.report]
    allow *
```

Note: some explanations of namespaceauth.conf are due in this documentation.

Processing Reports

As previously mentioned, by default the server stores incoming YAML reports to disk. There are other reports types available that can process each report as it arrives, or you can write a separate processor that handles the reports on your own schedule.

Using Builtin Reports

As with the rest of Puppet, you can configure the server to use different reports with either command-line arguments or configuration file changes. The value you need to change is called reports, and it must be a comma-separated list of the reports you want to use. Here's how you'd configure extra reports on the command line:

```
$ puppetmasterd —reports tagmail,store,log
```

Note that we're still specifying store here; any reports you specify replace the default, so you must still manually specify store if you want it. You can also specify none if you want the reports to just be thrown away.

Or we can include these configuration parameters in the configuration file, typically /etc/puppet/puppet.conf. For example:

```
#
#  /etc/puppet/puppet.conf
#
[puppetmasterd]
    reports = tagmail,store,log
```

Note that in the configuration file, the list of reports should be comma-separated and not enclosed in quotes (which is otherwise acceptable for a command-line invocation).

Writing Custom Reports

You can easily write your own report processor in place of any of the built-in reports. Just drop the report into lib/puppet/reports, using the existing reports as an example. This is only necessary on the server, as the report reciever does not run on the clients.

Using External Report Processors

Many people are only using the store report and writing an external report processor that processes many reports at once and produces summary pages. This is easiest if these processors are written in Ruby, since you can just read the YAML files in and de-serialize them into Ruby objects. Then, you can just do whatever you need with the report objects.

Available reports

Read the Report Reference for a list of available reports and how to configure them. It is automatically generated from the reports available in Puppet, and includes documentation on how to use each report.

Chapter 30

Getting Started With Puppet CloudPack

Learn how to install and start using CloudPack, Puppet's preview Faces extension for node bootstrapping.

Overview

Puppet CloudPack is a Puppet extension that adds new actions for creating and puppetizing new machines, especially Amazon AWS EC2 instances.

CloudPack gives you an easy command line interface to the following tasks:

- Create a new Amazon EC2 instance
- Install Puppet Enterprise on a remote machine of your choice
- Add a new puppet agent node to a Puppet Dashboard node group
- Do all of the above (plus sign the new node's certificate) with a single `puppet node bootstrap` invocation

Installing

To install Puppet CloudPack, simply clone the repository on your control node and add its lib directory to your `$RUBYLIB` or Ruby load path.

Prerequisites

Puppet CloudPack has several requirements beyond those of Puppet.

Software

CloudPack can only be used with **Puppet 2.7 or greater.** Classification of new nodes requires Puppet Dashboard 1.1.2 (unreleased at the time of this writing) or greater.

CloudPack also requires Fog, a Ruby cloud services library. You'll need to **ensure that Fog is installed** on the machine running CloudPack:

```
# gem install fog
```

Depending on your operating system and Ruby environment, you may need to manually install some of Fog's dependencies.

If you wish to use CloudPack to install Puppet on new nodes, you'll also need **a copy of the Puppet Enterprise universal tarball.** As of this writing, the distro-specific tarballs are not supported.

The machine running the CloudPack faces will need a working `/usr/bin/uuidgen` binary.

Services

Currently, Amazon EC2 is the only supported cloud platform for creating new machine instances; you'll need a pre-existing **Amazon EC2 account** to use this feature.

Configuration

Fog

For CloudPack to work, Fog needs to be configured with your AWS access key ID and secret access key. Create a ~/.fog file as follows:

```
:default:
  :aws_access_key_id:      XXXXXXXXXXXXXXXXXXXX
  :aws_secret_access_key: Xx+xxXX+XxxXXXXXXxxXxxXXXXxxxXXxXXxxxxXX
```

To test whether Fog is working, execute the following command:

```
$ ruby -rubygems -e 'require "fog"' -e 'puts Fog::Compute.new(:provider => "AWS").servers.length >= 0'
```

This should return "true"

If you do not have the ~/.fog configuration file correct, you may receive an error such as the following. In this case, please verify your aws_access_key_id and aws_secret_access_key are properly set in the ~/.fog file

```
fog-0.9.0/lib/fog/core/service.rb:155
in `validate_options': Missing required arguments: aws_access_key_id, aws_secret_access_key (ArgumentError
    )
        from /Users/jeff/.rvm/gems/ruby-1.8.7-p334@puppet/gems/fog-0.9.0/lib/fog/core/service.rb:53:in `
            new'
        from /Users/jeff/.rvm/gems/ruby-1.8.7-p334@puppet/gems/fog-0.9.0/lib/fog/compute.rb:13:in `new'
        from -e:2
```

EC2

Your EC2 account will need to have at least one **32-bit AMI of a supported Puppet Enterprise OS,**[1] at least one Amazon-managed **SSH keypair,** and a security group that **allows outbound traffic on port 8140 and SSH traffic from the machine running the CloudPack actions.** As of this writing, all of these resources **must be in the `us-east-1` region;** this will change in a later release of the CloudPack. We also hope to support 64-bit AMIs at a later date.

Your puppet master server will also have to be reachable from your newly created instances.

Provisioning

In order to use the `install` action, any newly provisioned instances will need to have their root user enabled, or will need a user account configured to `sudo` as root without a password.

puppet master

If you want to automatically sign certificates with the CloudPack, you'll have to allow the computer running the CloudPack actions to access the puppet master's `certificate_status` REST endpoint. This can be configured in the master's auth.conf file:

```
path /certificate_status
method save
auth yes
allow {certname}
```

If you're running the CloudPack actions on a machine other than your puppet master, you'll have to ensure it can communicate with the puppet master over port 8140 and your Puppet Dashboard server over port 3000.

Certificates and Keys

You'll also have to make sure the control node has a certificate signed by the puppet master's CA. If the control node is already known to the puppet master (e.g. it is or was a puppet agent node), you'll be able to use the existing certificate, but we recommend generating a per-user certificate for a more explicit and readable security policy. On the control node, run:

```
puppet certificate generate {certname} --ca-location remote
```

Then sign the certificate as usual on the master (puppet cert sign {certname}). On the control node again, run:

```
puppet certificate find ca --ca-location remote
puppet certificate find {certname} --ca-location remote
```

This should let you operate under the new certname when you run puppet commands with the –certname {certname} option.

The control node will also need a private key to allow SSH access to the new machine; for EC2 nodes, this is the private key from the keypair used to create the instance. If you are working with non-EC2 nodes, please note that the install action does not currently support keys with passphrases.

Installer Configuration

To install Puppet Enterprise on a node, you'll need a complete answers file to be read by the installer. See the PE documentation for more details. Note that the certname from the answers file is ignored, and the new instance will be given a UUID as its certname.

Usage

Puppet CloudPack provides five new actions on the node face:

- create: Creates a new EC2 machine instance.
- install: Install's Puppet Enterprise on an arbitrary machine, including non-cloud hardware.
- classify: Add a new node to a Puppet Dashboard node group.
- init: Perform the install and classify actions, and automatically sign the new agent node's certificate.
- bootstrap: Create a new EC2 machine instance and perform the init action on it.
- terminate: Tear down an EC2 machine instance.

puppet node create

Argument(s): none.

Options:

- --image, -i — The name of the AMI to use when creating the instance. **Required.**
- --keypair — The Amazon-managed SSH keypair to use for accessing the instance. **Required.**
- --group, -g, --security-group — The security group(s) to apply to the instance. Can be a single group or a path-separator (colon, on *nix systems) separated list of groups.

Example:

```
$ puppet node create --image ami-XxXXxXXX --keypair puppetlabs.admin
```

Creates a new EC2 machine instance, prints its SSH host key fingerprints, and returns its DNS name. If the process fails, Puppet will automatically clean up after itself and tear down the instance.

For security reasons, SSH fingerprints are obtained by observing the AWS console for the machine. This entails a noticeable wait, and the console output is sometimes not provided; if this happens, the instance will be kept alive and you will have to obtain host fingerprints through AWS.

puppet node install

Argument(s): the hostname of the system to install Puppet on.

Options:

- --login, -l, --username — The user to log in as. **Required.**
- --keyfile — The SSH private key file to use. This key cannot require a passphrase. **Required.**
- --installer-payload, --puppet — The location of the Puppet Enterprise universal tarball. **Required.**
- --installer-answers — The location of an answers file to use with the PE installer. **Required.**

Example:

```
puppet node install ec2-XXX-XXX-XXX-XX.compute-1.amazonaws.com \
--login root --keyfile ~/.ssh/puppetlabs-ec2_rsa \
--installer-payload ~/puppet-enterprise-1.0-all.tar.gz \
--installer-answers ~/pe-agent-answers
```

Install Puppet Enterprise on an arbitrary system and return the new agent node's certname. This action currently requires the universal PE tarball; per-distro tarballs are not supported.

Interactive installation is not supported, so you'll need an answers file. See the PE manual for complete documentation of the answers file format.

This action is not restricted to cloud machine instances, and will install PE on any machine accessible by SSH.

puppet node classify

Argument(s): the certname of the agent node to classify.

Options:

- --node-group, --as — The Puppet Dashboard node group to use. **Required.**
- --report_server — The hostname of your Puppet Dashboard server. Required unless properly configured in puppet.conf. This is a global Puppet option.
- --report_port — The port on which Puppet Dashboard is listening. Required unless properly configured in puppet.conf. This is a global Puppet option.
- --certname — The certname (Subject CN) of a certificate authorized by the puppet master to remotely sign CSRs. Required unless properly configured in puppet.conf. This is a global Puppet option.

Example:

```
puppet node classify ec2-XXX-XXX-XXX-XX.compute-1.amazonaws.com \
--as webserver_generic --report_server dashboard.puppetlabs.lan \
--report_port 3000 --certname cloud_admin
```

Make Puppet Dashboard aware of a newly created agent node and add it to a node group, thus allowing it to receive proper configurations on its next run. This action will have no material effect unless you're using Puppet dashboard for node classification.

This action is not restricted to cloud machine instances. It can be run multiple times for a single node.

puppet node init

Argument(s): the hostname of the system to install Puppet on.

Options: See “install” and “classify.”

Example:

```
puppet node init ec2-XXX-XXX-XXX-XX.compute-1.amazonaws.com \
--login root --keyfile ~/.ssh/puppetlabs-ec2_rsa \
--installer-payload ~/puppet-enterprise-1.0-all.tar.gz\
 --installer-answers ~/pe-agent-answers --as webserver_generic \
--report_server dashboard.puppetlabs.lan --report_port 3000 --certname cloud_admin
```

Install Puppet Enterprise on an arbitrary system (see “install”), classify it in Dashboard (see “classify”), and automatically sign its certificate request (using the certificate face’s sign action).

puppet node bootstrap

Argument(s): none.

Options: See “create,” “install,” and “classify.”

Example:

```
puppet node bootstrap --image ami-XxXXxXXX --keypair \
puppetlabs.admin --login root --keyfile ~/.ssh/puppetlabs-ec2_rsa \
--installer-payload ~/puppet-enterprise-1.0-all.tar.gz \
--installer-answers ~/pe-agent-answers --as webserver_generic \
--report_server dashboard.puppetlabs.lan --report_port 3000 \
--certname cloud_admin
```

Create a new EC2 machine instance and pass the new node’s hostname to the init action.

puppet node terminate

Argument(s): the hostname of the machine instance to tear down.

Options: none.

Example:

```
puppet node terminate init ec2-XXX-XXX-XXX-XX.compute-1.amazonaws.com
```

Tear down an EC2 machine instance.

1. Currently, the supported platforms for Puppet Enterprise are CentOS 5, RHEL 5, Debian 5, and Ubuntu 10.04 LTS.

Chapter 31

External Nodes

Traditionally, puppet master uses the node definitions in the main site manifest (site.pp) to choose which classes to apply to a node. But you can also classify nodes based on a pre-existing external data source, like an LDAP database or a set of flat files describing your infrastructure. Depending on the data you've collected, building an external node classifier (ENC) can be one of the easiest and most high-value ways to extend Puppet.

What Is an ENC?

An external node classifier is an executable that can be called by puppet master; it doesn't have to be written in Ruby. Its only argument is the name of the node to be classified, and it returns a YAML document describing the node.

Inside the ENC, you can reference any data source you want, including some of Puppet's own data sources, but from Puppet's perspective, it just puts in a node name and gets back a hash of information.

Considerations and Limitations

- The YAML returned by an ENC isn't an exact equivalent of a node definition in site.pp — it can't declare individual resources, declare relationships, or do conditional logic. The only things an ENC can do are **declare classes, assign top-scope variables, and set an environment.** This means an ENC is most effective if you've done a good job of separating your configurations out into classes and modules.
- Although ENCs can set an environment for a node, this is not very well supported — currently, the server-set environment will win during catalog compilation, but the client-set environment will win when downloading files. (See issue 3910 for more details.) We hope to make server-side environments work well in the future, but if you need them right now, the workaround is to use Puppet to manage puppet.conf on the agent and set the environment for the next run based on what the ENC thinks it should be.
- You can optionally combine an ENC with regular node definitions in site.pp. This works on the "I hope you brought enough for everybody" rule: things will work correctly if you have an ENC and no node definitions, but if there's at least one node definition, you need to have a default node defined or account for every node with a definition; Puppet will fail compilation with an error if a definition for a given node can't be found.
- Even if you aren't using node definitions, you can still use site.pp to do things like set global resource defaults.
- If an ENC doesn't produce any output and if the node name resembles a hostname, Puppet may call it again with a shortened version of the node name, successively removing higher-level domains and finally resorting to "default". *(This has not been tested recently, and may need updating.)* To suppress this behavior, turn on puppet master's strict_hostname_checking setting.

Connecting an ENC

To tell puppet master to use an ENC, you need to set two configuration options: node_terminus has to be set to "exec", and external_nodes should have the path to the executable.

```
[master]
  node_terminus = exec
  external_nodes = /usr/local/bin/puppet_node_classifier
```

ENC Output Format

There have been three versions of the ENC output format.

Puppet 2.6.5 and Higher

ENCs MUST return either a YAML hash or nothing. This hash MAY contain classes, parameters, and environment keys, and MUST contain at least either classes or parameters. ENCs SHOULD exit with an exit code of 0 when functioning normally, and MAY exit with a non-zero exit code if you wish puppet master to behave as though the requested node was not found.

Classes

If present, the value of classes MUST be either an array of class names or a hash whose keys are class names. That is, the following are equivalent:

```
classes:
  - common
  - puppet
  - dns
  - ntp

classes:
  common:
  puppet:
  dns:
  ntp:
```

Parameterized classes cannot be used with the array syntax. When using the hash key syntax, the value for a parameterized classe SHOULD be a hash of the class's attributes and values. Each value MAY be a string, number, array, or hash. Non-parameterized classes MAY have empty values.

```
classes:
    common:
    puppet:
    ntp:
        ntpserver: 0.pool.ntp.org
    aptsetup:
        additional_apt_repos:
            - deb localrepo.magpie.lan/ubuntu lucid production
            - deb localrepo.magpie.lan/ubuntu lucid vendor
```

Parameters

If present, the value of the parameters key MUST be a hash of valid variable names and associated values; these will be exposed to the compiler as top scope variables. Each value MAY be a string, number, array, or hash.

```
parameters:
    ntp_servers:
        - 0.pool.ntp.org
        - ntp.puppetlabs.lan
    mail_server: mail.puppetlabs.lan
    iburst: true
```

Environment

If present, the value of environment MUST be a string representing the desired environment for this node. As noted above, ENC-set environments are not currently reliable, although this can be worked around by managing puppet.conf as a resource.

```
environment: production
```

Complete Example

```
classes:
    common:
    puppet:
    ntp:
        ntpserver: 0.pool.ntp.org
    aptsetup:
        additional_apt_repos:
            - deb localrepo.magpie.lan/ubuntu lucid production
            - deb localrepo.magpie.lan/ubuntu lucid vendor
parameters:
    ntp_servers:
        - 0.pool.ntp.org
        - ntp.puppetlabs.lan
    mail_server: mail.puppetlabs.lan
    iburst: true
environment: production
```

Puppet 0.23.0 through 2.6.4

As above, with the following exception:

Classes

If present, the value of classes MUST be an array of class names. Parameterized classes cannot be used with an ENC.

Puppet 0.22.4 and Lower

ENCs MUST return two lines of text, separated by a newline (LF). The first line MUST be the name of a parent node defined in the main site manifest. The second line MUST be a space-separated list of classes. ENCs MUST exit with exit code 0; Puppet's behavior when faced with a non-zero ENC exit code is undefined.

Complete example

```
basenode
common puppet dns ntp
```

Tricks, Notes, and Further Reading

- Although only the node name is directly passed to an ENC, it can make decisions based on other facts about the node by querying the inventory service REST API or using the puppet facts subcommand shipped with Puppet 2.7.
- Puppet's "exec" node_terminus is just one way for Puppet to build node objects, and it's optimized for flexibility and for the simplicity of its API. There are situations where it can make more sense to design a native node terminus instead of an ENC, one example being the "ldap" node terminus that ships with Puppet. See the LDAP nodes documentation on the wiki for more info.

Chapter 32

Inventory Service

Set up and begin using the inventory service with one or more puppet master servers. **This document refers to a feature currently under development.**

Puppet 2.6.7 adds support for maintaining, reading, and searching an inventory of nodes. This can be used to generate reports about the composition of your site, to drastically extend the capabilities of your external node classifier, and probably to do a lot of things we haven't even thought of yet. This service is designed as a *hackable public API.*

Why

In order to compile and serve a catalog to an agent node, the puppet master has to collect a large amount of information about that node, in the form of Facter facts. If that info is written to a persistent store whenever it's collected, it suddenly becomes a fairly detailed inventory of every node that Puppet controls or has controlled at a given site! This can be tremendously useful: Imagine being able to instantly find out which computers are still running CentOS 4.5 and need to be upgraded, or which computers have less than a certain amount of physical memory, or what percentage of your current infrastructure is in the cloud on EC2 instances. Build a good enough interface to the inventory, and the data becomes *knowledge.* That knowledge can then drive other tools; for example, you could let your provisioning system or node classifier make decisions about new hardware based on the properties of the existing infrastructure.

Several users have built custom inventory functionality by directly reading either the puppet master's YAML fact cache or the optional storeconfigs database. But both of these approaches were non-optimal:

- The YAML cache is strictly local to one puppet master, and isn't an accurate inventory in multi-master environments. Furthermore, repeatedly deserializing YAML is terribly slow, which can cause real problems depending on the use case. (Searching by fact, in particular, is basically not an option.)
- Storeconfigs, on the other hand, is global to the site, but it's essentially a private API: Since the only officially supported use of it is for sharing exported resources, the only way to get fact data out of it is to read the database directly, and there's been no guarantee against the schema changing. Furthermore, storeconfigs is too heavyweight for users who just want an inventory, since it stores every resource and tag in each node's catalog in addition to the node's facts, and even the "thin" storeconfigs option stores a LOT of data. Implementing storeconfigs at a reasonable scale demands setting up a message queue, and even that extra infrastructure doesn't necessarily make it viable at a very large scale.

Thus, the Puppet inventory service: a relatively speedy implementation that does one thing well and exposes a public network API.

What It Is

The *inventory* is a collection of node facts. The *inventory service* is a retrieval, storage, and search API exposed to the network by the puppet master.

The puppet master updates the inventory when agent nodes report their facts, which happens every time puppet agent requests a catalog. Optionally, additional puppet masters can use the REST API to send facts from their agents to the central inventory.

Other tools, including Puppet Dashboard, can query the inventory via the puppet master's REST API. An API call can return:

- Complete facts for a single node

or

- A list of nodes whose facts meet some search condition

Information in the inventory is never automatically expired, but it is timestamped.

Consumers of the Inventory Service

The inventory service is primarily meant for external applications, and its data is not currently read by any part of Puppet. The only application which currently consumes the inventory data is Puppet Dashboard version 1.1.0, which can display facts in node views and provides a web interface for searching the inventory by fact.

Using the Inventory Service

The inventory service is plain vanilla REST: Submit HTTP requests, get back structured fact or host data.

To read from the inventory, submit secured HTTP requests to the puppet master's facts and facts_search REST endpoints in the appropriate environment. Your API client will have to have an SSL certificate signed by the puppet master's CA.

Full documentation of these endpoints can be found here, but a summary follows:

- To retrieve the facts for testnode.localdomain, send a GET request to https://puppet:8140/production/facts/testn
- To retrieve a list of all Ubuntu nodes with two or more processors, send a GET request to https://puppet:8140/proc

In both cases, be sure to specify an Accept: pson or Accept: yaml header.

Setting Up the Inventory Service

Configuring the Inventory Backend

The inventory service's backend is configured with the facts_terminus setting in the puppet master's section of puppet.conf.

For Prototyping: YAML

```
[master]
    facts_terminus = yaml
```

You can actually start using the inventory service with the YAML backend immediately — yaml is the default value for facts_terminus, and the YAML cache of any previously used puppet master will already be populated with fact information. Just configure access (see below) and you're good to go.

For Production: Database

```
[master]
    facts_terminus = inventory_active_record
    dblocation = {sqlite file path (sqlite only)}
    dbadapter = {sqlite3|mysql|postgresql|oracle_enhanced}
    dbname = {database name (all but sqlite)}
    dbuser = {database user (all but sqlite)}
    dbpassword = {database password (all but sqlite)}
    dbserver = {database server (MySQL and PostgreSQL only)}
    dbsocket = {database socket file (MySQL only; optional)}
```

Before using the database facts backend, you'll have to fulfill a number of requirements:

- Puppet master will need access to both a database and a user account with all privileges on that database; setting that up is outside the scope of this document. The database server can be remote or on the local host.
- You'll need to ensure that the copy of Ruby in use by puppet master is able to communicate with your chosen type of database server. This will *always* entail ensuring that Rails is installed, and will *likely* require installing a specific Ruby library to interface with the database (e.g. the libmysql−ruby package on Debian and Ubuntu or the mysql gem on other operating systems).

These requirements are essentially identical to those used by storeconfigs, so the Puppet wiki page for storeconfigs can be helpful. Getting MySQL on the local host configured is very well-documented; other options, less so.

For Multiple Puppet Masters: REST

```
[master]
    facts_terminus = rest
    inventory_server = {inventorying puppet master; defaults to "puppet"}
    inventory_port = 8140 (unless changed)
```

In addition to writing to its local YAML cache, any puppet master with a facts_terminus of rest will submit facts to another puppet master, which is hopefully using the inventory_active_record backend.

Configuring Access

By default, the inventory service is not accessible! This is sane. The inventory service exposes sensitive information about your infrastructure over the network, so you'll need to carefully control access with the `rest_authconfig` (a.k.a. `auth.conf`) file.

For prototyping your inventory application on a scratch puppet master, you can just permit all access to the facts endpoint:

```
path /facts
auth any
method find, search
allow *
```

(Note that this will allow access to both facts and facts_search, since the path is read as a prefix.)

For production deployment, you'll need to allow find and search access for your application, allow save access for any other puppet masters at your site (so they can submit their nodes' facts), and deny access to all other machines. (Since agent nodes submit their facts as part of their request to the catalog resource, they don't require access to the facts or facts_search resources.) One such possible ACL set would be:

```
path /facts
auth yes
method find, search
allow custominventorybrowser.puppetlabs.lan

path /facts
auth yes
method save
allow puppetmaster1.puppetlabs.lan, puppetmaster2.puppetlabs.lan, puppetmaster3.puppetlabs.lan
```

Configuring Certificates

To connect your application securely, you'll need a certificate signed by your site's puppet CA. There are two main ways to get this:

- **On the puppet master:**
  - Run `puppet cert --generate {certname for application}`.
  - Then, retrieve the private key (`{ssldir}/certs/{certname}.pem`) and the signed certificate (`{ssldir}/private_keys/{certname}.pem`) and move them to your application server.
- **Manually:**
  - Generate an RSA private key: `openssl genrsa -out {certname}.pem 1024`.
  - Generate a certificate signing request (CSR): `openssl req -new -key {certname}.pem -subj "/CN={certname}" -out request.csr`.
  - Submit the CSR to the puppet master for signing: `curl -k -X PUT -H "Content-Type: text/plain" --data-binary @request.csr https://puppet:8140/production/certificate_request/no_key`.
  - Sign the certificate on the puppet master: `puppet cert --sign {certname}`.
  - Retrieve the certificate: `curl -k -H -o {certname}.pem "Accept: s" https://puppet:8140/production/certificate/{certname}`

For one-off applications, generating it on the master is obviously easier, but if you're building a tool for distribution elsewhere, your users will appreciate it if you script the manual method and emulate the way puppet agent gets a cert.

Protect your application's private key appropriately, since it's the gateway to your inventory data.

In the event of a security breach, the application's certificate is revokable the same way any puppet agent certificate would be.

Testing the Inventory Service

On a machine that you've authorized to access the facts and facts_search resources, you can test the API using `curl`, as described in the REST API docs. To retrieve facts for a node:

```
curl -k -H "Accept: yaml" https://puppet:8140/production/facts/{node certname}
```

To insert facts for a fictional node into the inventory:

```
curl -k -X PUT -H 'Content-Type: text/yaml' --data-binary @/var/lib/puppet/yaml/facts/hostname.yaml https://puppet:8140/production/facts/{node certname}
```

To find out which nodes at your site are Intel Macs:

```
curl -k -H "Accept: pson" https://puppet:8140/production/facts_search/search?facts.hardwaremodel=i386&facts.kernel=Darwin
```

Chapter 33

Plugins in Modules

Learn how to distribute custom facts and types from the server to managed clients automatically.

Details

This page describes the deployment of custom facts and types for use by the client via modules.

Custom types and facts are stored in modules. These custom types and facts are then gathered together and distributed via a file mount on your Puppet master called plugins.

This technique can also be used to bundle functions for use by the server when the manifest is being compiled. Doing so is a two step process which is described further on in this document.

To enable module distribution you need to make changes on both the Puppet master and the clients.

Note: Plugins in modules is supported in 0.24.x onwards and modifies the pluginsync model supported in releases prior to 0.24.x. It is NOT supported in earlier releases of Puppet but may be present as a patch in some older Debian Puppet packages. The older 0.24.x configuration for plugins in modules is documented at the end of this page.

Module structure for 0.25.x and later

In Puppet version 0.25.x and later, plugins are stored in the lib directory of a module, using an internal directory structure that mirrors that of the Puppet code:

```
{modulepath}
 {module}
     lib
         facter
         puppet
             parser
                 functions
             provider
                 exec
                 package
                 etc... (any resource type)
             type
```

As the directory tree suggests, custom facts should go in lib/facter/, custom types should go in lib/puppet/type/, custom providers should go in lib/puppet/provider/{type}/, and custom functions should go in lib/puppet/parser/functions/.

For example:

A custom user provider:

{modulepath}/{module}/lib/puppet/provider/user/custom_user.rb

A custom package provider:

{modulepath}/{module}/lib/puppet/provider/package/custom_pkg.rb

A custom type for bare Git repositories:

{modulepath}/{module}/lib/puppet/type/gitrepo.rb

A custom fact for the root of all home directories (that is, /home on Linux, /Users on Mac OS X, etc.):

{modulepath}/{module}/lib/facter/homeroot.rb

And so on.

Most types and facts should be stored in which ever module they are related to; for example, a Bind fact might be distributed in your Bind module. If you wish to centrally deploy types and facts you could create a separate module just for this purpose, for example one called custom. This module needs to be a valid module (with the correct directory structure and an init.pp file).

So, if we are using our custom module and our modulepath is /etc/puppet/modules then types and facts would be stored in the following directories:

```
/etc/puppet/modules/custom/lib/puppet/type
/etc/puppet/modules/custom/lib/puppet/provider
/etc/puppet/modules/custom/lib/puppet/parser/functions
/etc/puppet/modules/custom/lib/facter
```

Note: 0.25.x versions of Puppet have a known bug whereby plugins are instead loaded from the deprecated plugins directories of modules when applying a manifest locally with the puppet command, even though puppetmasterd will correctly serve the contents of lib/ directories to agent nodes. This bug is fixed in Puppet 2.6.

Enabling Pluginsync

After setting up the directory structure, we then need to turn on pluginsync in our puppet.conf configuration file on both the master and the clients:

```
[main]
pluginsync = true
```

Note on Usage for Server Custom Functions

Functions are executed on the server while compiling the manifest. A module defined in the manifest can include functions in the plugins directory. The custom function will need to be placed in the proper location within the manifest first:

{modulepath}/{module}/lib/puppet/parser/functions

Note that this location is not within the puppetmaster's $libdir path. Placing the custom function within the module plugins directory will not result in the puppetmasterd loading the new custom function. The puppet client can be used to help deploy the custom function by copying it from modulepath/module/lib/puppet/-parser/functions to the proper $libdir location. To do so run the puppet client on the server. When the client runs it will download the custom function from the module's lib directory and deposit it within the correct location in $libdir. The next invocation of the Puppet master by a client will autoload the custom function.

As always custom functions are loaded once by the Puppet master. Simply replacing a custom function with a new version will not cause Puppet master to automatically reload the function. You must restart the Puppet master.

Legacy 0.24.x and Plugins in Modules

For older Puppet release the `lib` directory was called `plugins`.

So for types you would place them in:

```
{modulepath}/{module}/plugins/puppet/type
```

For providers you place them in:

```
{modulepath}/{module}/plugins/puppet/provider
```

Similarly, Facter facts belong in the facter subdirectory of the library directory:

```
{modulepath}/{module}/plugins/facter
```

If we are using our custom module and our modulepath is /etc/puppet/modules then types and facts would be stored in the following directories:

```
/etc/puppet/modules/custom/plugins/puppet/type
/etc/puppet/modules/custom/plugins/puppet/provider
/etc/puppet/modules/custom/plugins/facter
```

Enabling pluginsync for 0.24.x versions

For 0.24.x versions you may need to specify some additional options:

```
[main]
pluginsync=true
factsync=true
factpath = $vardir/lib/facter
```

Chapter 34

Custom Facts

Extend facter by writing your own custom facts to provide information to Puppet.

Adding Custom Facts to Facter

Sometimes you need to be able to write conditional expressions based on site-specific data that just isn't available via Facter (or use a variable in a template that isn't there). A solution can be achieved by adding a new fact to Facter. These additional facts can then be distributed to Puppet clients and are available for use in manifests.

The Concept

You can add new facts by writing a snippet of Ruby code on the Puppet master. We then use Plugins In Modules to distribute our facts to the client.

An Example

Let's say we need to get the output of uname -i to single out a specific type of workstation. To do these we create a fact. We start by giving the fact a name, in this case, hardware_platform, and create our new fact in a file, hardware_platform.rb, on the Puppet master server:

```
# hardware_platform.rb

Facter.add("hardware_platform") do
        setcode do
                %x{/bin/uname -i}.chomp
        end
end
```

Note that the chomp is required to provide clean data.

We then use the instructions in Plugins In Modules page to copy our new fact to a module and distribute it. During your next Puppet run the value of our new fact will be available to use in your manifests.

The best place to get ideas about how to write your own custom facts is to look at the existing Facter fact code. You will find lots of examples of how to interpret different types of system data and return useful facts.

You may not be able to view your custom fact when running facter on the client node. If you are unable to view the custom fact, try adding the "factpath" to the FACTERLIB environmental variable:

```
export FACTERLIB=/var/lib/puppet/lib/facter
```

Using other facts

You can write a fact which uses other facts by accessing Facter.value("somefact") or simply Facter.somefact. The former will return nil for unknown facts, the latter will raise an exception. An example:

```
Facter.add("osfamily") do
    setcode do
        begin
            Facter.lsbdistid
        rescue
            Facter.loadfacts()
        end
        distid = Facter.value('lsbdistid')
        if distid.match(/RedHatEnterprise|CentOS|Fedora/)
            family = "redhat"
        elsif distid == "ubuntu"
            family = "debian"
        else
            family = distid
        end
        family
    end
end
```

Here it is important to note that running facter myfact on the command line will not load other facts, hence the above code calls Facter.loadfacts to work in this mode, too. loadfacts will only load the default facts.

To still test your custom puppet facts, which are usually only loaded by puppetd, there is a small hack:

```
mkdir rubylib
cd rubylib
ln -s /path/to/puppet/facts facter
RUBYLIB=. facter
```

Testing

Of course, we can test that our code works before adding it to Puppet.

Create a directory called facter/ somewhere (we often use ~/lib/ruby/facter), and set the environment variable $RUBYLIB to its parent directory. You can then run facter, and it will import your code:

```
$ mkdir -p ~/lib/ruby/facter ; export RUBYLIB=~/lib/ruby
$ cp /path/to/hardware_platform.rb $RUBYLIB/facter
$ facter hardware_platform
SUNW,Sun-Blade-1500
```

Adding this path to your $RUBYLIB also means you can see this fact when you run Puppet. Hence, you should now see the following when running puppetd:

```
# puppetd -vt --factsync
info: Retrieving facts
info: Loading fact hardware_platform
...
```

Alternatively, you can set $FACTERLIB to a directory with your new facts in, and they will be recognised on the Puppet master.

It is important to note that to use the facts on your clients you will still need to distribute them using the Plugins In Modules method.

Viewing Fact Values

You can also determine what facts (and their values) your clients return by checking the contents of the client's yaml output. To do this we check the $yamldir (by default $vardir/yaml/) on the Puppet master:

```
# grep kernel /var/lib/puppet/yaml/node/puppetslave.example.org.yaml
  kernel: Linux
  kernelrelease: 2.6.18-92.el5
  kernelversion: 2.6.18
```

Legacy Fact Distribution

For Puppet versions prior to 0.24.0:

On older versions of Puppet, prior to 0.24.0, a different method called factsync was used for custom fact distribution. Puppet would look for custom facts on puppet://$server/facts by default and you needed to run puppetd with --factsync option (or add factsync = true to puppetd.conf). This would enable the syncing of these files to the local file system and loading them within puppetd.

Facts were synced to a local directory ($vardir/facts, by default) before facter was run, so they would be available the first time. If $factsource was unset, the --factsync option is equivalent to:

```
file { $factdir: source => "puppet://puppet/facts", recurse => true }
```

After the facts were downloaded, they were loaded (or reloaded) into memory.

Some additional options were avaialble to configure this legacy method:

The following command line or config file options are available (default options shown):

- factpath ($vardir/facts): Where Puppet should look for facts. Multiple directories should be colon-separated, like normal PATH variables. By default, this is set to the same value as factdest, but you can have multiple fact locations (e.g., you could have one or more on NFS).
- factdest ($vardir/facts): Where Puppet should store facts that it pulls down from the central server.
- factsource (puppet://$server/facts): From where to retrieve facts. The standard Puppet file type is used for retrieval, so anything that is a valid file source can be used here.
- factsync (false): Whether facts should be synced with the central server.
- factsignore (.svn CVS): What files to ignore when pulling down facts.

Remember the approach described above for factsync is now deprecated and replaced by the plugin approach described in the Plugins In Modules page.

Chapter 35

Custom Functions

Extend the Puppet interpreter by writing your own custom functions.

Writing your own functions

The Puppet language and interpreter is very extensible. One of the places you can extend Puppet is in creating new functions to be executed on the puppet master at the time that the manifest is compiled. To give you an idea of what you can do with these functions, the built-in template and include functions are implemented in exactly the same way as the functions you're learning to write here.

Custom functions are written in Ruby, so you'll need a working understanding of the language before you begin.

Gotchas

There are a few things that can trip you up when you're writing your functions:

- Your function will be executed on the server. This means that any files or other resources you reference must be available on the server, and you can't do anything that requires direct access to the client machine.
- There are actually two completely different types of functions available — *rvalues* (which return a value) and *statements* (which do not). If you are writing an rvalue function, you must pass `:type => :rvalue` when creating the function; see the examples below.
- The name of the file containing your function must be the same as the name of function; otherwise it won't get automatically loaded.
- To use a *fact* about a client, use `lookupvar('{fact name}')` instead of `Facter['{fact name}'].value`. See examples below.

Where to put your functions

Functions are implemented in individual .rb files (whose filenames must match the names of their respective functions), and should be distributed in modules. Put custom functions in the lib/puppet/parser/functions subdirectory of your module; see Plugins in Modules for additional details (including compatibility with versions of Puppet prior to 0.25.0).

If you are using a version of Puppet prior to 0.24.0, or have some other compelling reason to not use plugins in modules, functions can also be loaded from .rb files in the following locations:

- `$libdir/puppet/parser/functions`
- `puppet/parser/functions` sub-directories in your Ruby `$LOAD_PATH`

First Function — small steps

New functions are defined by executing the newfunction method inside the Puppet::Parser::Functions module. You pass the name of the function as a symbol to newfunction, and the code to be run as a block. So a trivial function to write a string to a file in /tmp might look like this:

```
module Puppet::Parser::Functions
  newfunction(:write_line_to_file) do |args|
    filename = args[0]
    str = args[1]
    File.open(args[0], 'a') {|fd| fd.puts str }
  end
end
```

To use this function, it's as simple as using it in your manifest:

```
write_line_to_file('/tmp/some_file', "Hello world!")
```

(Note that this is not a useful function by any stretch of the imagination.)

The arguments to the function are passed into the block via the `args` argument to the block. This is simply an array of all of the arguments given in the manifest when the function is called. There's no real parameter validation, so you'll need to do that yourself.

This simple `write_line_to_file` function is an example of a *statement* function. It performs an action, and does not return a value. The other type of function is an *rvalue* function, which you must use in a context which requires a value, such as an `if` statement, a `case` statement, or a variable or attribute assignment. You could implement a `rand` function like this:

```
module Puppet::Parser::Functions
  newfunction(:rand, :type => :rvalue) do |args|
    rand(vals.empty? ? 0 : args[0])
  end
end
```

This function works identically to the Ruby built-in rand function. Randomising things isn't quite as useful as you might think, though. The first use for a `rand` function that springs to mind is probably to vary the minute of a cron job. For instance, to stop all your machines from running a job at the same time, you might do something like:

```
cron { run_some_job_at_a_random_time:
  command => "/usr/local/sbin/some_job",
  minute => rand(60)
}
```

But the problem here is quite simple: every time the Puppet client runs, the rand function gets re-evaluated, and your cron job moves around. The moral: just because a function *seems* like a good idea, don't be so quick to assume that it'll be the answer to all your problems.

Using Facts and Variables

Which raises the question: what *should* you do if you want to splay your cron jobs on different machines? The trick is to tie the minute value to something that's invariant in time, but different across machines. Perhaps the MD5 hash of the hostname, modulo 60, or maybe the IP address of the host converted to an integer, modulo 60. Neither guarantees uniqueness, but you can't really expect that with a range of no more than 60 anyway.

But given that functions are run on the puppet master, how do you get at the hostname or IP address of the agent node? The answer is that facts returned by facter can be used in our functions.

Example 1

```
require 'ipaddr'

module Puppet::Parser::Functions
  newfunction(:minute_from_address, :type => :rvalue) do |args|
```

```
    IPAddr.new(lookupvar('ipaddress')).to_i % 60
  end
end
```

Example 2

```
require 'md5'

module Puppet::Parser::Functions
  newfunction(:hour_from_fqdn, :type => :rvalue) do |args|
    MD5.new(lookupvar('fqdn')).to_s.hex % 24
  end
end
```

Basically, to get a fact's or variable's value, you just call lookupvar('{fact name}').

Calling Functions from Functions

Functions can be accessed from other functions by prefixing them with function_.

Example

```
module Puppet::Parser::Functions
  newfunction(:myfunc2, :type => :rvalue) do |args|
    function_myfunc1(...)
  end
end
```

Handling Errors

To throw a parse/compile error in your function, in a similar manner to the fail() function:

```
raise Puppet::ParseError, "my error"
```

Troubleshooting Functions

If you're experiencing problems with your functions loading, there's a couple of things you can do to see what might be causing the issue:

1 - Make sure your function is parsing correctly, by running:

```
ruby -rpuppet my_funct.rb
```

This should return nothing if the function is parsing correctly, otherwise you'll get an exception which should help troubleshoot the problem.

2 - Check that the function is available to Puppet:

```
irb
> require 'puppet'
> require '/path/to/puppet/functions/my_funct.rb'
> Puppet::Parser::Functions.function(:my_funct)
=> "function_my_funct"
```

Substitute :my_funct with the name of your function, and it should return something similar to "function_my_funct " if the function is seen by Puppet. Otherwise it will just return false, indicating that you still have a problem (and you'll more than likely get a "Unknown Function" error on your clients).

Referencing Custom Functions In Templates

To call a custom function within a Puppet Template, you can do:

```
<%= scope.function_namegoeshere(["one","two"]) %>
```

Replace "namegoeshere" with the function name, and even if there is only one argument, still include the array brackets.

Notes on Backward Compatibility

Accessing Files With Older Versions of Puppet

In Puppet 2.6.0 and later, functions can access files with the expectation that it will just work. In versions prior to 2.6.0, functions that accessed files had to explicitly warn the parser to recompile the configuration if the files they relied on changed.

If you find yourself needing to write custom functions for older versions of Puppet, the relevant instructions are preserved below.

Accessing Files in Puppet 0.23.2 through 0.24.9

Until Puppet 0.25.0, safe file access was achieved by adding `self.interp.newfile($filename)` to the function. E.g., to accept a file name and return the last line of that file:

```
module Puppet::Parser::Functions
  newfunction(:file_last_line, :type => :rvalue) do |args|
    self.interp.newfile(args[0])
    lines = IO.readlines(args[0])
    lines[lines.length - 1]
  end
end
```

Accessing Files in Puppet 0.25.x

In release 0.25.0, the necessary code changed to:

```
parser = Puppet::Parser::Parser.new(environment)
parser.watch_file($filename)
```

This new code was used identically to the older code:

```
module Puppet::Parser::Functions
  newfunction(:file_last_line, :type => :rvalue) do |args|
    parser = Puppet::Parser::Parser.new(environment)
    parser.watch_file($filename)
    lines = IO.readlines(args[0])
    lines[lines.length - 1]
  end
end
```

Chapter 36

Custom Types

Learn how to create your own custom types & providers in Puppet

---

Organizational Principles

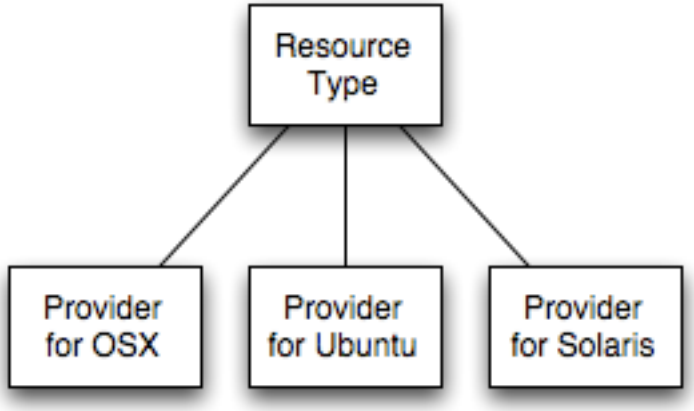

Figure 36.1: image

When creating a new Puppet type, you will be create two things: The resource type itself, which we normally just call a 'type', and the provider(s) for that type. While Puppet does not require Ruby experience to use, extending Puppet with new Puppet types and providers does require some knowledge of the Ruby programming language, as is the case with new functions and facts. If you're new to Ruby, what is going on should still be somewhat evident from the examples below, and it is easy to learn.

The resource types provide the model for what you can do; they define what parameters are present, handle input validation, and they determine what features a provider can (or should) provide.

The providers implement support for that type by translating calls in the resource type to operations on the system. As mentioned in our Introduction and language guide, an example would be that "yum" and "apt" are both different providers that fulfill the "package" type.

Deploying Code

Once you have your code, you will need to have it both on the server and also distributed to clients.

The best place to put this content is within Puppet's configured libdir. The libdir is special because you can use the pluginsync system to copy all of your plugins from the fileserver to all of your clients (and seperate Puppetmasters, if they exist)). To enable pluginsync, set pluginsync=true in puppet.conf and, if necessary, set the pluginsource setting. The contents of pluginsource will be copied directly into libdir, so make sure you make a puppet/type directory in your pluginsource, too.

In Puppet 0.24 and later, the "old" pluginsync function has been deprecated and you should see the Plugins In Modules page for details of distributing custom types and facts via modules.

The internals of how types are created have changed over Puppet's lifetime, and this document will focus on best practices, skipping over all the things you can but probably shouldn't do.

Resource Types

When defining the resource type, focus on what the resource can do, not how it does it (that is the job for providers!).

The first thing you have to figure out is what properties the resource has. Properties are the changeable bits, like a file's owner or a user's UID.

After adding properties, Then you need to add any other necessary parameters, which can affect how the resource behaves but do not directly manage the resource itself. Parameters handle things like whether to recurse when managing files or where to look for service init scripts.

Resource types also support special parameters, called MetaParameters, that are supported by all resource types, but you can safely ignore these since they are already defined and you won't normally add more. You may remember that things like require are metaparameters.

Types are created by calling the newtype method on Puppet::Type, with the name of the type as the only required argument. You can optionally specify a parent class; otherwise, Puppet::Type is used as the parent class. You must also provide a block of code used to define the type:

You may wish to read up on "Ruby blocks" to understand more about the syntax. Blocks are a very powerful feature of Ruby and are not surfaced in most programming languages.

```
Puppet::Type.newtype(:database) do
    @doc = "Create a new database."
    ... the code ...
end
```

The above code should be stored in puppet/type/database.rb (within the libpath), because of the name of the type we're creating ("database").

A normal type will define multiple properties and possibly some parameters. Once these are defined, as long as the type is put into lib/puppet/type anywhere in Ruby's search path, Puppet will autoload the type when you reference it in the Puppet language.

We have already mentioned Puppet provides a libdir setting where you can copy the files outside the Ruby search path. See also Plugins In Modules

All types should also provide inline documention in the @doc class instance variable. The text format is in Restructured Text.

Properties

Here's where we define how the resource really works. In most cases, it's the properties that interact with your resource's providers. If you define a property named owner, then when you are retrieving the state of your resource, then the owner property will call the owner method on the provider. In turn, when you are setting the state (because the resource is out of sync), then the owner property will call the owner= method to set the state on disk.

There's one common exception to this: The ensure property is special because it's used to create and destroy resources. You can set this property up on your resource type just by calling the ensurable method in your type definition:

```
Puppet::Type.newtype(:database) do
    ensurable
    ...
end
```

This property uses three methods on the provider: create, destroy, and exists?. The last method, somewhat obviously, is a boolean to determine if the resource current exists. If a resource's ensure property is out of sync, then no other properties will be checked or modified.

You can modify how ensure behaves, such as by adding other valid values and determining what methods get called as a result; see existing types like package for examples.

The rest of the properties are defined a lot like you define the types, with the newproperty method, which should be called on the type:

```
Puppet::Type.newtype(:database) do
    ensurable
    newproperty(:owner) do
        desc "The owner of the database."
        ...
    end
end
```

Note the call to desc; this sets the documentation string for this property, and for Puppet types that get distributed with Puppet, it is extracted as part of the Type reference.

When Puppet was first developed, there would normally be a lot of code in this property definition. Now, however, you normally only define valid values or set up validation and munging. If you specify valid values, then Puppet will only accept those values, and it will automatically handle accepting either strings or symbols. In most cases, you only define allowed values for ensure, but it works for other properties, too:

```
newproperty(:enable) do
    newvalue(:true)
    newvalue(:false)
end
```

You can attach code to the value definitions (this code would be called instead of the property= method), but it's normally unnecessary.

For most properties, though, it is sufficient to set up validation:

```
newproperty(:owner) do
    validate do |value|
        unless value =~ /^\w+/
            raise ArgumentError, "%s is not a valid user name" % value
        end
    end
end
```

Note that the order in which you define your properties can be important: Puppet keeps track of the definition order, and it always checks and fixes properties in the order they are defined.

Customizing Behaviour

By default, if a property is assigned multiple values in an array, it is considered in sync if any of those values matches the current value. If, instead, the property should only be in sync if all values match the current value (e.g., a list of times in a cron job), you can declare this:

```
newproperty(:minute, :array_matching => :all) do # defaults to :first
    ...
end
```

You can also customize how information about your property gets logged. You can create an is_to_s method to change how the current values are described, should_to_s to change how the desired values are logged, and change_to_s to change the overall log message for changes. See current types for examples.

Handling Property Values

Handling values set on properties is currently somewhat confusing, and will hopefully be fixed in the future. When a resource is created with a list of desired values, those values are stored in each property in its @should instance variable. You can retrieve those values directly by calling should on your resource (although note that when array_matching is set to first you get the first value in the array, otherwise you get the whole array):

```
myval = should(:color)
```

When you're not sure (or don't care) whether you're dealing with a property or parameter, it's best to use value:

```
myvalue = value(:color)
```

Parameters

Parameters are defined essentially exactly the same as properties; the only difference between them is that parameters never result in methods being called on providers.

Like ensure, one parameter you will always want to define is the one used for naming the resource. This is nearly always called name:

```
newparam(:name) do
    desc "The name of the database."
end
```

You can name your naming parameter something else, but you must declare it as the namevar:

```
newparam(:path, :namevar => true) do
    ...
end
```

In this case, path and name are both accepted by Puppet, and it treats them equivalently.

If your parameter has a fixed list of valid values, you can declare them all at once:

```
newparam(:color) do
    newvalues(:red, :green, :blue, :purple)
end
```

You can specify regexes in addition to literal values; matches against regexes always happen after equality comparisons against literal values, and those matches are not converted to symbols. For instance, given the following definition:

```
newparam(:color) do
    desc "Your color, and stuff."

    newvalues(:blue, :red, /.+/)
end
```

If you provide blue as the value, then your parameter will get set to :blue, but if you provide green, then it will get set to "green".

Validation and Munging

If your parameter does not have a defined list of values, or you need to convert the values in some way, you can use the validate and munge hooks:

```
newparam(:color) do
    desc "Your color, and stuff."

    newvalues(:blue, :red, /.+/)

    validate do |value|
        if value == "green"
            raise ArgumentError,
                "Everyone knows green databases don't have enough RAM"
        else
```

```
            super
        end
    end

    munge do |value|
        case value
        when :mauve, :violet # are these colors really any different?
            :purple
        else
            super
        end
    end
end
```

The default validate method looks for values defined using newvalues and if there are any values defined it accepts only those values (this is exactly how allowed values are validated). The default munge method converts any values that are specifically allowed into symbols. If you override either of these methods, note that you lose this value handling and symbol conversion, which you'll have to call super for.

Values are always validated before they're munged.

Lastly, validation and munging *only** happen when a value is assigned. They have no role to play at all during use of a given value, only during assignment.

Automatic Relationships

Your type can specify automatic relationships it can have with resources. You use the autorequire hook, which requires a resource type as an argument, and your code should return a list of resource names that your resource could be related to:

```
autorequire(:file) do
  ["/tmp", "/dev"]
end
```

Note that this won't throw an error if resources with those names do not exist; the purpose of this hook is to make sure that if any required resources are being managed, they get applied before the requiring resource.

Providers

Look at the Provider Development page for intimate detail; this document will only cover how the resource types and providers need to interact.Because the properties call getter and setter methods on the providers, except in the case of ensure, the providers must define getters and setters for each property.

Provider Features

A recent development in Puppet (around 0.22.3) is the ability to declare what features providers can have. The type declares the features and what's required to make them work, and then the providers can either be tested for whether they suffice or they can declare that they have the features. Additionally, individual properties and parameters in the type can declare that they require one or more specific features, and Puppet will throw an error if those prameters are used with providers missing those features:

```
newtype(:coloring) do
    feature :paint, "The ability to paint.", :methods => [:paint]
    feature :draw, "The ability to draw."

    newparam(:color, :required_features => %w{paint}) do
        ...
    end
end
```

The first argument to the feature method is the name of the feature, the second argument is its description, and after that is a hash of options that help Puppet determine whether the feature is available. The only option currently supported is specifying one or more methods that must be defined on the provider. If no methods are specified, then the provider needs to specifically declare that it has that feature:

```
Puppet::Type.type(:coloring).provide(:drawer) do
    has_feature :draw
end
```

The provider can specify multiple available features at once with has_features.

When you define features on your type, Puppet automatically defines a bunch of class methods on the provider:

- feature?: Passed a feature name, will return true if the feature is available or false otherwise.
- features: Returns a list of all supported features on the provider.
- satisfies?: Passed a list of feature, will return true if they are all available, false otherwise.

Additionally, each feature gets a separate boolean method, so the above example would result in a paint? method on the provider.

Chapter 37

Complete Resource Example

This document walks through the definition of a very simple resource type and one provider. We'll build the resource up slowly, and the provider along with it. See Custom Types and Provider Development for more information on the individual classes. As with creating Custom Facts and Custom Functions, these examples involve Ruby programming.

Resource Creation

Nearly every resource needs to be able to be created and destroyed, and resources have to have names, so we'll start with those two features. Puppet's property support has a helper method called ensurable that handles modeling creation and destruction; it creates an ensure property and adds absent and present values for it, which in turn require three methods on the provider, create, destroy, and exists?. Here's the first start to the resource. We're going to create one called 'file' — this is an example of how we'd create a resource for something Puppet already has. You can see how this would be extensible to handle one of your own ideas:

```
Puppet::Type.newtype(:file) do
    @doc = "Manage a file (the simple version)."

    ensurable

    newparam(:name) do
        desc "The full path to the file."
    end
end
```

Here we have provided the resource type name (it's file), a simple documentation string (which should be in Restructured Text format), a parameter for the name of the file, and we've used the ensurable method to say that our file is both createable and destroyable.

To see how we would use this on the provider side, let's look at a simple provider:

```
Puppet::Type.type(:file).provide(:posix) do
    desc "Normal Unix-like POSIX support for file management."

    def create
        File.open(@resource[:name], "w") { |f| f.puts "" } # Create an empty file
    end

    def destroy
        File.unlink(@resource[:name])
    end

    def exists?
        File.exists?(@resource[:name])
    end
end
```

Here you can see that the providers use a different way of specifying their documentation, which is not something that has been unified in Puppet yet.

In addition to the docs and the provider name, we provide the three methods that the ensure property requires. You can see that in this case we're just using Ruby's built-in File abilities to create an empty file, remove the file, or test whether the file exists.

Let's enhance our resource somewhat by adding the ability to manage the file mode. Here's the code we need to add to the resource:

```
newproperty(:mode) do
    desc "Manage the file's mode."

    defaultto "640"
end
```

Notice that we're specifying a default value, and that it is a string instead of an integer (file modes are in octal, and most of us are used to specifying integers in decimal). You can pass a block to defaultto instead of a value, if you don't have a simple value. (For more about blocks, see the Ruby language documentation).

Here's the code we need to add to the provider to understand modes:

```
def create
    File.open(@resource[:name], "w") { |f| f.puts "" } # Create an empty file
    # Make sure the mode is correct
    should_mode = @resource.should(:mode)
    unless self.mode == should_mode
        self.mode = should_mode
    end
end

# Return the mode as an octal string, not as an integer.
def mode
    if File.exists?(@resource[:name])
        "%o" % (File.stat(@resource[:name]).mode & 007777)
    else
        :absent
    end
end

# Set the file mode, converting from a string to an integer.
def mode=(value)
    File.chmod(Integer("0" + value), @resource[:name])
end
```

Note that the getter method returns the value, it doesn't attempt to modify the resource itself. Also, when the setter gets passed the value it is supposed to set; it doesn't attempt to figure out the appropriate value to use. This should always be true of how providers are implemented.

Also notice that the ensure property, when created by the ensurable method, behaves differently because it uses methods for creation and destruction of the file, whereas normal properties use getter and setter methods. When a resource is being created, Puppet expects the create method (or, actually, any changes done within ensure) to make any other necessary changes. This is because most often resources are created already configured correctly, so it doesn't make sense for Puppet to test it manually (e.g., useradd support is set up to add all specified properties when useradd is run, so usermod doesn't need to be run afterward).

You can see how the absent and present values are defined by looking in the property.rb file; here's the most important snippet:

```
newvalue(:present) do
    if @resource.provider and @resource.provider.respond_to?(:create)
        @resource.provider.create
    else
        @resource.create
    end
    nil # return nil so the event is autogenerated
end

newvalue(:absent) do
    if @resource.provider and @resource.provider.respond_to?(:destroy)
```

```
        @resource.provider.destroy
    else
        @resource.destroy
    end
    nil # return nil so the event is autogenerated
end
```

There are a lot of other options in creating properties, parameters, and providers, but this should provide a decent starting point.

See Also

- Provider Development
- Creating Custom Types

Chapter 38

Provider Development

Information about writing providers to provide implementation for types.

About

The core of Puppet's cross-platform support is via Resource Providers, which are essentially back-ends that implement support for a specific implementation of a given resource type. For instance, there are more than 20 package providers, including providers for package formats like dpkg and rpm along with high-level package managers like apt and yum. A provider's main job is to wrap client-side tools, usually by just calling out to those tools with the right information.

Not all resource types have or need providers, but any resource type concerned about portability will likely need them.

We will use the apt and dpkg package providers as examples throughout this document, and the examples used are current as of 0.23.0.

Declaration

Providers are always associated with a single resource type, so they are created by calling the provide class method on that resource type. When declarating a provider, you can specify a parent class — for instance, all package providers have a common parent class:

```
Puppet::Type.type(:package).provide :dpkg, :parent => Puppet::Provider::Package do
    desc "..."
    ...
end
```

Note the call desc there; it sets the documentation for this provider, and should include everything necessary for someone to use this provider.

Providers can also specify another provider (from the same resource type) as their parent:

```
Puppet::Type.type(:package).provide :apt, :parent => :dpkg, :source => :dpkg do
    ...
end
```

Note that we're also specifying that this provider uses the dpkg source; this tells Puppet to deduplicate packages from dpkg and apt, so the same package does not show up in an instance list from each provider type. Puppet defaults to creating a new source for each provider type, so you have to specify when a provider subclass shares a source with its parent class.

Suitability

The first question to ask about a new provider is where it will be functional, which Puppet describes as suitable. Unsuitable providers cannot be used to do any work, although we're working on making the suitability test late-binding, meaning that you could have a resource in your configuration that made a provider suitable. If you start puppetd or puppet in debug mode, you'll see the results of failed provider suitability tests for the resource types you're using.

Puppet providers include some helpful class-level methods you can use to both document and declare how to determine whether a given provider is suitable. The primary method is commands, which actually does two things for you: It declares that this provider requires the named binary, and it sets up class and instance methods with the name provided that call the specified binary. The binary can be fully qualified, in which case that specific path is required, or it can be unqualified, in which case Puppet will find the binary in the shell path and use that. If the binary cannot be found, then the provider is considered unsuitable. For example, here is the header for the dpkg provider (as of 0.23.0):

```
commands :dpkg => "/usr/bin/dpkg"
commands :dpkg_deb => "/usr/bin/dpkg-deb"
commands :dpkgquery => "/usr/bin/dpkg-query"
```

In addition to looking for binaries, Puppet can compare Facter facts, test for the existence of a file, or test whether a given value is true or false. For file extistence, truth, or false, just call the confine class method with exists, true, or false as the name of the test and your test as the value:

```
confine :exists => "/etc/debian_release"
confine :true => Puppet.features.rrd?
confine :false => Puppet.features.rails?
```

To test Facter values, just use the name of the fact:

```
confine :operatingsystem => [:debian, :solaris]
confine :puppetversion => "0.23.0"
```

Note that case doesn't matter in the tests, nor does it matter whether the values are strings or symbols. It also doesn't matter whether you specify an array or a single value — Puppet does an OR on the list of values.

Default Providers

Providers are generally meant to be hidden from the users, allowing them to focus on resource specification rather than implementation details. Toward this end, Puppet does what it can to choose an appropriate default provider for each resource type.

This is generally done by a single provider declaring that it is the default for a given set of facts, using the defaultfor class method. For instance, this is the apt provider's declaration:

```
defaultfor :operatingsystem => :debian
```

The same fact matching functionality is used, so again case does not matter.

Provider/Resource API

Providers never do anything on their own; all of their action is triggered through an associated resource (or, in special cases, from the transaction). Because of this, resource types are essentially free to define their own provider interface if necessary, and providers were initially developed without a clear resource/provider API (mostly because it wasn't clear whether such an API was necessary or what it would look like). At this point, however, there is a default interface between the resource type and the provider.

This interface consists entirely of getter and setter methods. When the resource is retrieving its current state, it iterates across all of its properties and calls the getter method on the provider for that property. For instance, when a user resource is having its state retrieved and its uid and shell properties are being managed, then the resource will call uid and shell on the provider to figure out what the current state of each of those properties is. This method call is in the retrieve method in Puppet::Property.

When a resource is being modified, it calls the equivalent setter method for each property on the provider. Again using our user example, if the uid was in sync but the shell was not, then the resource would call shell=(value) with the new shell value.

The transaction is responsible for storing these returned values and deciding which value to actually send, and it does its work through a PropertyChange instance. It calls sync on each of the properties, which in turn just call the setter by default.

You can override that interface as necessary for your resource type, but in the hopefully-near future this API will become more solidified.

Note that all providers must define an instances class method that returns a list of provider instances, one for each existing instance of that provider. For instance, the dpkg provider should return a provider instance for every package in the dpkg database.

Provider Methods

By default, you have to define all of your getter and setter methods. For simple cases, this is sufficient — you just implement the code that does the work for that property.

However, because things are rarely so simple, Puppet attempts to help in a few ways.

Prefetching

First, Puppet transactions will prefetch provider information by calling prefetch on each used provider type. This calls the instances method in turn, which returns a list of provider instances with the current resource state already retrieved and stored in a @property_hash instance variable. The prefetch method then tries to find any matching resources, and assigns the retrieved providers to found resources. This way you can get information on all of the resources you're managing in just a few method calls, instead of having to call all of the getter methods for every property being managed. Note that it also means that providers are often getting replaced, so you cannot maintain state in a provider.

Resource Methods

For providers that directly modify the system when a setter method is called, there's no substitute for defining them manually. But for resources that get flushed to disk in one step, such as the ParsedFile providers, there is a mk_resource_methods class method that creates a getter and setter for each property on the resource. These methods just retrieve and set the appropriate value in the @property_hash variable.

Flushing

Many providers model files or parts of files, so it makes sense to save all of the writes up and do them in one run. Providers in need of this functionality can define a flush instance method to do this. The transaction will call this method after all values are synced (which means that the provider should have them all in its @property_hash variable) but before refresh is called on the resource (if appropriate).

Chapter 39

Using Puppet From Source

Puppet is implemented in Ruby and uses standard Ruby libraries. You should be able to run Puppet on any Unix-style host with ruby. Windows support is planned for future releases.

Before you Begin

Make sure your host has Ruby version 1.8.2 or later:

```
$ ruby -v
```

and, if you want to run the tests, rake:

```
$ rake -V
```

While Puppet should work with ruby 1.8.1, there have been many reports of problems with this version.

Make sure you have Git:

```
$ git --version
```

Get the Source

Puppet relies on another Puppet Labs library, Facter. Create a working directory and get them both:

```
$ SETUP_DIR=~/git
$ mkdir -p $SETUP_DIR
$ cd $SETUP_DIR
$ git clone git://github.com/puppetlabs/facter
$ git clone git://github.com/puppetlabs/puppet
```

You will need to periodically run:

```
$ git pull --rebase origin
```

From your repositories to periodically update your clone to the latest code.

If you want access to all of the tags in the git repositories, so that you can compare releases, for instance, do the following from within the repository:

```
$ git fetch --tags
```

Then you can compare two releases with something like this:

```
$ git diff 0.25.1 0.25.2
```

Most of the development on puppet is done in branches based either on features or the major revision lines. Currently the “stable” branch is 0.25.x and development is in the “master” branch. You can change to and track branches by using the following:

```
git checkout --track -b 0.25.x origin/0.25.x
```

Tell Ruby How to Find Puppet and Facter

Finally, we need to put the puppet binaries into our path and make the Puppet and Facter libraries available to Ruby:

```
$ PATH=$PATH:$SETUP_DIR/facter/bin:$SETUP_DIR/puppet/bin
$ RUBYLIB=$SETUP_DIR/facter/lib:$SETUP_DIR/puppet/lib
$ export PATH RUBYLIB
```

Note: environment variables (depending on your OS) can get stripped when running as sudo. If you experience problems, you may want to simply execute things as root.

Next we must install facter. Facter changes far less often than Puppet and is a very minimal tool/library:

```
$ cd facter
$ sudo ruby ./install.rb
```

Chapter 40

Development Lifecycle

If you'd like to work on Puppet and submit a contribution, we'd be glad to have you.

Since this information changes often, please see the Puppet Wiki for the latest details.

Chapter 41

REST API

Both puppet master and puppet agent have RESTful API's that they use to communicate. The basic structure of the url to access this API is

```
https://yourpuppetmaster:8140/{environment}/{resource}/{key}
https://yourpuppetclient:8139/{environment}/{resource}/{key}
```

Details about what resources are available and the formats they return are below.

REST API Security

Puppet usually takes care of security and SSL certificate management for you, but if you want to use the RESTful API outside of that you'll need to manage certificates yourself when you connect. This can be done by using a pre-existing signed agent certificate, by generating and signing a certificate on the puppet master and manually distributing it to the connecting host, or by re-implementing puppet agent's generate / submit signing request / received signed certificate behavior in your custom app.

The security policy for the REST API can be controlled through the `rest_authconfig` file. For testing purposes, it is also possible to permit unauthenticated connections from all hosts or a subset of hosts; see the `rest_authconfig` documentation for more details.

Testing the REST API using curl

An example of how you can use the REST API to retrieve the catalog for a node can be seen using curl.

```
curl --cert /etc/puppet/ssl/certs/mymachine.pem --key /etc/puppet/ssl/private_keys/mymachine.pem --cacert
    /etc/puppet/ssl/ca/ca_crt.pem -H 'Accept: yaml' https://puppetmaster:8140/production/catalog/
    mymachine
```

Most of this command consists of pointing curl to the appropriate SSL certificates, which will be different depending on your ssldir location and your node's certname. For simplicity and brevity, future invocations of curl will be provided in insecure mode, which is specified with the `-k` or `--insecure` flag. Insecure connections can be enabled for one or more nodes in the `rest_authconfig` file. The above curl invocation without certificates would be as follows:

```
curl --insecure -H 'Accept: yaml' https://puppetmaster:8140/production/catalog/mymachine
```

Basically we just send a header specifying the format or formats we want back, and the RESTful URI for getting a catalog for mymachine in the production environment. Here's a snippet of the output you might get back:

```
--- &id001 !ruby/object:Puppet::Resource::Catalog
  aliases: {}
    applying: false
      classes: []
      ...
```

Another example to get back the CA Certificate of the puppetmaster doesn't require you to be authenticated with your own signed SSL Certificates, since that's something you would need before you authenticate.

```
curl —insecure –H 'Accept: s' https://puppetmaster:8140/production/certificate/ca

———BEGIN CERTIFICATE———
MIICHTCCAYagAwIBAgIBATANBgkqhkiG9w0BAQUFADAXMRUwEwYDVQQDDAxwdXBw
```

The master and agent shared API

Resources

Returns a list of resources, like executing puppet resource (ralsh) on the command line.

GET /{environment}/resource/{resource_type}/{resource_name}

GET /{environment}/resources/{resource_type}

Example:

```
curl –k –H "Accept: yaml" https://puppetmaster:8140/production/resource/user/puppet
curl –k –H "Accept: yaml" https://puppetclient:8139/production/resources/user
```

Certificate

Get a certficate or the master's CA certificate.

GET /certificate/{ca, other}

Example:

```
curl –k –H "Accept: s" https://puppetmaster:8140/production/certificate/ca
curl –k –H "Accept: s" https://puppetclient:8139/production/certificate/puppetclient
```

The master REST API

A valid and signed certificate is required to retrieve these resources.

Catalogs

Get a catalog from the node.

GET /{environment}/catalog/{node certificate name}

Example:

```
curl –k –H "Accept: pson" https://puppetmaster:8140/production/catalog/myclient
```

Certificate Revocation List

Get the certificate revocation list.

GET /certificate_revocation_list/ca

Example:

```
curl –k –H "Accept: s" https://puppetmaster:8140/production/certificate_revocation_list/ca
```

Certificate Request

Retrieve or save certificate requests.

GET /{environment}/certificate_requests/no_key

GET /{environment}/certificate_request/{node certificate name}

PUT /{environment}/certificate_request/no_key

Example:

```
curl -k -H "Accept: yaml" https://puppetmaster:8140/production/certificate_requests/all
curl -k -H "Accept: yaml" https://puppetmaster:8140/production/certificate_request/{agent certname}
curl -k -X PUT -H "Content-Type: text/plain" --data-binary @cert_request.csr https://puppetmaster:8140/
    production/certificate_request/no_key
```

To manually generate a CSR from an existing private key:

```
openssl req -new -key private_key.pem -subj "/CN={node certname}" -out request.csr
```

The subject can only include a /CN=, nothing else. Puppet master will determine the certname from the body of the cert, so the request can be pointed to any key for this endpoint.

Certificate Status

Puppet 2.7.0 and later.

Read or alter the status of a certificate or pending certificate request. This endpoint is roughly equivalent to the puppet cert command; rather than returning complete certificates, signing requests, or revocation lists, this endpoint returns information about the various certificates (and potential and former certificates) known to the CA.

GET /{environment}/certificate_status/{certname}

Retrieve a PSON hash containing information about the specified host's certificate. Similar to puppet cert --list {certname}.

GET /{environment}/certificate_statuses/no_key

Retrieve a list of PSON hashes containing information about all known certificates. Similar to puppet cert --list --all.

PUT /{environment}/certificate_status/{certname}

Change the status of the specified host's certificate. The desired state is sent in the body of the PUT request as a one-item PSON hash; the two allowed complete hashes are {"desired_state":"signed"} (for signing a certificate signing request; similar to puppet cert --sign) and {"desired_state":"revoked"} (for revoking a certificate; similar to puppet cert --revoke); see examples below for details.

When revoking certificates, you may wish to use a DELETE request instead, which will also clean up other info about the host.

DELETE /{environment}/certificate_status/{hostname}

Cause the certificate authority to discard all SSL information regarding a host (including any certificates, certificate requests, and keys). This **does not** revoke the certificate if one is present; if you wish to emulate the behavior of puppet cert --clean, you must PUT a desired_state of revoked before deleting the host's SSL information.

Examples:

```
curl -k -H "Accept: pson" https://puppetmaster:8140/production/certificate_status/testnode.localdomain
curl -k -H "Accept: pson" https://puppetmaster:8140/production/certificate_statuses/all
curl -k -X PUT -H "Content-Type: text/pson" --data '{"desired_state":"signed"}' https://puppetmaster:8140/
    production/certificate_status/client.network.address
curl -k -X PUT -H "Content-Type: text/pson" --data '{"desired_state":"revoked"}' https://puppetmaster
    :8140/production/certificate_status/client.network.address
curl -k -X DELETE -H "Accept: pson" https://puppetmaster:8140/production/certificate_status/client.network
    .address
```

Reports

Submit a report.

PUT /{environment}/report/{node certificate name}

Example:

```
curl –k –X PUT –H "Content–Type: text/yaml" –d "{key:value}" https://puppetclient:8139/production/report/
    puppetclient
```

Resource Types

Return a list of resources from the master

GET /{environment}/resource_type/{hostclass,definition,node}

GET /{environment}/resource_types/*

Example:

```
curl –k –H "Accept: yaml" https://puppetmaster:8140/production/resource_type/puppetclient
curl –k –H "Accept: yaml" https://puppetmaster:8140/production/resource_types/*
```

File Bucket

Get or put a file into the file bucket.

GET /{environment}/file_bucket_file/md5/{checksum}

PUT /{environment}/file_bucket_file/md5/{checksum}

GET /{environment}/file_bucket_file/md5/{checksum}?diff_with={checksum} (diff 2 files; **Puppet 2.6.5 and later**)

HEAD /{environment}/file_bucket_file/md5/{checksum} (determine if a file is present; **Puppet 2.6.5 and later**)

Examples:

```
curl –k –H "Accept: s" https://puppetmaster:8140/production/file_bucket_file/md5/
    e30d4d879e34f64e33c10377e65bbce6
curl –k –X PUT –H "Content–Type: text/plain" Accept: s" https://puppetmaster:8140/production/
    file_bucket_file/md5/e30d4d879e34f64e33c10377e65bbce6 —data–binary @foo.txt
curl –k –H "Accept: s" https://puppetmaster:8140/production/file_bucket_file/md5/
    e30d4d879e34f64e33c10377e65bbce6?diff_with=6572b5dc4c56366aaa36d996969a8885
curl –k –I –H "Accept: s" https://puppetmaster:8140/production/file_bucket_file/md5/
    e30d4d879e34f64e33c10377e65bbce6
```

File Server

Get a file from the file server.

GET /file_{metadata, content}/{file}

File serving is covered in more depth on the wiki

Node

Returns the Puppet::Node information (including facts) for the specified node

GET /{environment}/node/{node certificate name}

Example:

```
curl –k –H "Accept: yaml" https://puppetmaster:8140/production/node/puppetclient
```

Status

Just used for testing

GET /{environment}/status/no_key

Example:

```
curl -k -H "Accept: pson" https://puppetmaster:8140/production/status/puppetclient
```

Facts

GET /{environment}/facts/{node certname}

```
curl -k -H "Accept: yaml" https://puppetmaster:8140/production/facts/{node certname}
```

PUT /{environment}/facts/{node certname}

```
curl -k -X PUT -H 'Content-Type: text/yaml' --data-binary @/var/lib/puppet/yaml/facts/hostname.yaml https
    ://localhost:8140/production/facts/{node certname}
```

Facts Search

GET /{environment}/facts_search/search?{facts search string}

```
curl -k -H "Accept: pson" https://puppetmaster:8140/production/facts_search/search?facts.processorcount.ge
    =2&facts.operatingsystem=Ubuntu
```

Facts search strings are constructed as a series of terms separated by &; if there is more than one term, the search combines the terms with boolean AND. There is currently no API for searching with boolean OR. Each term is composed as follows:

```
facts.{name of fact}.{comparison type}={string for comparison}
```

If you leave off the .{comparison type}, the comparison will default to simple equality. The following comparison types are available:

String/general comparison

- eq — == (default)
- ne — !=

Numeric comparison

- lt — <
- le — <=
- gt — >
- ge — >=

The agent REST API

By default, puppet agent is set not to listen to HTTP requests. To enable this you must set listen = true in the puppet.conf or pass --listen true to puppet agent when starting. Due to a known bug in the 2.6.x releases of Puppet, puppet agent will not start with listen = true unless a namespaceauth.conf file exists, even though this file is not consulted. The node's rest_authconfig file must also allow access to the agent's resources, which isn't permitted by default.

Facts

GET /{environment}/facts/no_key

Example:

```
curl –k –H "Accept: yaml" https://puppetclient:8139/production/facts/no_key
```

Run

Cause the client to update like puppetrun or puppet kick

PUT /{environment}/run/no_key

Example:

```
curl –k –X PUT –H "Content–Type: text/pson" –d "{}" https://puppetclient:8139/production/run/no_key
```

Chapter 42

Language Guide

The purpose of Puppet's language is to make it easy to specify the resources you need to manage on the machines you're managing.

This guide will show you how the language works, going through some basic concepts. Understanding the Puppet language is key, as it's the main driver of how you tell your Puppet managed machines what to do.

Ready To Dive In?

Puppet language is really relatively simple compared to many programming languages. As you are reading over this guide, it may also be helpful to look over various Puppet modules people have already written. Complete real world examples can serve as a great introduction to Puppet. See the Modules page for more information and some links to list of community developed Puppet content.

Language Feature by Release

Feature 0.23.1 0.24.6 0.24.7 0.25.0 2.6.0 Plusignment operator (+>) X X X X X Multiple Resource relationships X X X X Class Inheritance Overrides X X X X Appending to Variables (+=) X X X X Class names starting with 0–9 X X X X Multi-line C-style comments X X X Node regular expressions X X Expressions in Variables X X RegExes in conditionals X X Elsif in conditionals X Chaining Resources X Hashes X Parameterised Class X Run Stages X The "in" syntax X

Resources

The fundamental unit of modelling in Puppet is a resource. Resources describe some aspect of a system; it might be a file, a service, a package, or perhaps even a custom resource that you have developed. We'll show later how resources can be aggregated together with "defines" and "classes", and even show how to organize things with "modules", but resources are what we should start with first.

Each resource has a type, a title, and a list of attributes — each resource in Puppet can support various attributes, though many of them will have reasonable defaults and you won't have to specify all of them.

You can find all of the supported resource types, their valid attributes, and documentation for all of it in the References.

Let's get started. Here's a simple example of a resource in Puppet, where we are describing the permissions and ownership of a file:

```
file { '/etc/passwd':
  owner => 'root',
  group => 'root',
  mode  => '0644',
}
```

Any machine on which this snippet is executed will use it to verify that the passwd file is configured as specified.

The field before the colon is the resource's *title,* which must be unique and can be used to refer to the resource in other parts of the Puppet configuration. Following the title are a series of *attributes* and their *values.*

Most resources have an attribute (often called simply name) whose value will default to the title if you don't specify it. (Internally, this is called the "namevar.") For the file type, the path will default to the title. A resource's namevar value almost always has to be unique. (The exec and notify types are the exceptions.)

For simple resources that don't vary much, leaving out the name or path and falling back to the title is sufficient. But for resources with long names, or in cases where filenames differ between operating systems, it makes more sense to choose a symbolic title:

```
file { 'sshdconfig':
  path => $operatingsystem ? {
    solaris => '/usr/local/etc/ssh/sshd_config',
    default => '/etc/ssh/sshd_config',
  },
  owner => 'root',
  group => 'root',
  mode  => '0644',
}
```

This makes it easy to refer to the file resource elsewhere in our configuration, since the title is always the same.

For instance, let's add a service that depends on the file:

```
service { 'sshd':
  subscribe => File['sshdconfig'],
}
```

This will cause the sshd service to get restarted when the sshdconfig file changes. You'll notice that when we reference a resource we capitalise the name of the resource, for example File[sshdconfig]. When you see an uppercase resource type, that's always a reference. A lowercase version is a declaration. Since resources can only be declared once, repeating the same declaration twice will cause an error. This is an important feature of Puppet that makes sure your configuration is well modelled.

What happens if our resource depends on multiple resources? From Puppet version 0.24.6 you can specify multiple relationships like so:

```
service { 'sshd':
  require => File['sshdconfig', 'sshconfig', 'authorized_keys']
}
```

It's important to note here that the title alone identifies the resource. Even if the resource seems to conceptually point to the same entity, it's the title that matters. The following is possible in Puppet, but is to be avoided as it can lead to errors once things get sent down to the client.

```
file { 'sshdconfig':
  name  => '/usr/local/etc/ssh/sshd_config',
  owner => 'root',
}

file { '/usr/local/etc/ssh/sshd_config':
  owner => 'sshd',
}
```

Metaparameters

In addition to the attributes specific to each Resource Type Puppet also has global attributes called metaparameters. Metaparameters are parameters that work with any resource type.

In the examples in the section above we used two metaparameters, subscribe and require, both of which build relationships between resources. You can see the full list of all metaparameters in the Metaparameter Reference, though we'll point out additional ones we use as we continue the guide.

Resource Defaults

Sometimes you will need to specify a default parameter value for a set of resources; Puppet provides a syntax for doing this, using a capitalized resource specification that has no title. For instance, in the example below, we'll set the default path for all execution of commmands:

```
Exec { path => '/usr/bin:/bin:/usr/sbin:/sbin' }
exec { 'echo this works': }
```

The first statement in this snippet provides a default value for exec resources; Exec resources require either fully qualified paths or a path in which to look for the executable. Individual resources can still override this path when needed, but this saves typing. This way you can specify a single default path for your entire configuration, and then override that value as necessary.

Defaults work with any resource type in Puppet.

Defaults are not global — they only affect the current scope and scopes below the current one. If you want a default setting to affect your entire configuration, your only choice currently is to specify them outside of any class. We'll mention classes in the next section.

Resource Collections

Aggregation is a powerful concept in Puppet. There are two ways to combine multiple resources into one easier to use resource: Classes and defined resource types. Classes model fundamental aspects of nodes, they say "this node IS a webserver" or "this node is one of these". In programming terminology classes are singletons — they only ever get evaluated once per node.

Defined resource types, on the other hand, can be reused many times on the same node. They essentially work as if you created your own Puppet type just by using the language. They are meant to be evaluated multiple times, with different inputs each time. This means you can pass variable values into the defines.

Both classes and defines are very useful and you should make use of them when building out your puppet infrastructure.

Classes

Classes are introduced with the class keyword, and their contents are wrapped in curly braces. The following simple example creates a simple class that manages two separate files:

```
class unix {
  file {
    '/etc/passwd':
      owner => 'root',
      group => 'root',
      mode  => '0644';
    '/etc/shadow':
      owner => 'root',
      group => 'root',
      mode  => '0440';
  }
}
```

You'll notice we introduced some shorthand here. This is the same as saying:

```
class unix {
  file { '/etc/passwd':
    owner => 'root',
    group => 'root',
    mode  => '0644',
  }
  file { '/etc/shadow':
    owner => 'root',
    group => 'root',
    mode  => '0440',
  }
}
```

Classes also support a simple form of object inheritance. For those not acquainted with programming terms, this means that we can extend the functionality of the previous class without copy/pasting the entire class. Inheritance allows subclasses to override resource settings defined in parent classes. A class can only inherit from one other class, not more than one. In programming terms, this is called 'single inheritance'.

```
class freebsd inherits unix {
  File['/etc/passwd'] { group => 'wheel' }
  File['/etc/shadow'] { group => 'wheel' }
}
```

If we needed to undo some logic specified in a parent class, we can use undef like so:

```
class freebsd inherits unix {
  File['/etc/passwd'] { group => undef }
}
```

In the above example, nodes which include the unix class will have the password file's group set to root, while nodes including freebsd would have the password file group ownership left unmodified.

In Puppet version 0.24.6 and higher, you can specify multiple overrides like so:

```
class freebsd inherits unix {
  File['/etc/passwd', '/etc/shadow'] { group => 'wheel' }
}
```

There are other ways to use inheritance. In Puppet 0.23.1 and higher, it's possible to add values to resource parameters using the '+>' ('plusignment') operator:

```
class apache {
  service { 'apache': require => Package['httpd'] }
}

class apache-ssl inherits apache {
  # host certificate is required for SSL to function
  Service['apache'] { require +> File['apache.pem'] }
}
```

The above example makes the second class require all the packages in the first, with the addition of 'apache.pem'.

To append multiple requires, use array brackets and commas:

```
class apache {
  service { 'apache': require => Package['httpd'] }
}

class apache-ssl inherits apache {
  Service['apache'] { require +> [ File['apache.pem'], File['/etc/httpd/conf/httpd.conf'] ] }
}
```

The above would make the require parameter in the apache-ssl class equal to

```
[Package['httpd'], File['apache.pem'], File['/etc/httpd/conf/httpd.conf']]
```

Like resources, you can also create relationships between classes with 'require', like so:

```
class apache {
  service { 'apache': require => Class['squid'] }
}
```

The above example uses the require metaparameter to make the apache class dependent on the squid class.

In Puppet version 0.24.6 and higher, you can specify multiple relationships like so:

```
class apache {
  service { 'apache':
    require => Class['squid', 'xml', 'jakarta'],
  }
}
```

It's not dangerous to reference a class with a require more than once. Classes are evaluated using the include function (which we will mention later). If a class has already been evaluated once, then include essentially does nothing.

Parameterised Classes

In Puppet release 2.6.0 and later, classes are extended to allow the passing of parameters into classes.

To create a class with parameters you can now specify:

```
class apache($version) {
  ... class contents ...
}
```

Classes with parameters are not declared using the include function but with an alternate syntax similar to a resource declaration:

```
node webserver {
  class { 'apache': version => '1.3.13' }
}
```

You can also specify default parameter values in your class like so:

```
class apache($version = '1.3.13', $home = '/var/www') {
  ... class contents ...
}
```

Run Stages

Run stage were added in Puppet version 2.6.0, you now have the ability to specify any number of stages which provide another method to control the ordering of resource management in puppet. If you have a large number of resources in your catalog it may become tedious and cumbersome to explicitly manage every relationship between the resources where order is important. In this situation, run-stages provides you the ability to associate a class to a single stage. Puppet will guarantee stages run in a specific predictable order every catalog run.

In order to use run-stages, you must first declare additional stages beyond the already present main stage. You can then configure puppet to manage each stage in a specific order using the same resource relationship syntax, before, require, "->" and "<-". The relationship of stages will then guarantee the ordering of classes associated with each stage.

By default there is only one stage named "main" and all classes are automatically associated with this stage. Unless explicitly stated, a class will be associated with the main stage. With only one stage the effect of run stages is the same as previous versions of puppet since resources within a stage are managed in arbitrary order unless they have explicit relationships declared.

In order to declare additional stage resources, follow the same consistent and simple declarative syntax of the puppet language:

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

All classes associated with the first stage are to be managed before the classes associated with the main stage. All classes associated with the last stage are to be managed after the classes associated with the main stage.

Once stages have been declared, a class may be associated with a stage other than main using the "stage" class parameter.

```
class {
  'apt-keys': stage => first;
  'sendmail': stage => main;
  'apache':   stage => last;
}
```

Associate all resources in the class apt-keys with the first run stage, all resources in the class sendmail with the main stage, and all resources in the apache class with the last stage.

This short declaration guarantees resources in the apt-keys class are managed before resources in the sendmail class, which in turn is managed before resources in the apache class.

Please note that stage is not a metaparameter. The run stage must be specified as a class parameter and as such classes must use the resource declaration syntax as shown rather than the "include" statement.

Defined Resource Types

Defined resource types follow the same basic form as classes, but they are introduced with the define keyword (not class) and they support arguments but no inheritance. As mentioned previously, defined resource types take parameters and can be reused multiple times on the same system. Suppose we want to create a resource collection that creates source control repositories. We probably would want to create multiple repositories on the same system, so we would use a defined type, not a class. Here's an example:

```
define svn_repo($path) {
  exec { "/usr/bin/svnadmin create ${path}/${title}":
    unless => "/bin/test -d ${path}",
  }
}

svn_repo { 'puppet_repo': path => '/var/svn_puppet' }
svn_repo { 'other_repo':  path => '/var/svn_other' }
```

Note how parameters specified in the definition (define svn_repo($path)) must appear as resource attributes (path => '/var/svn_puppet') whenever a resource of the new type is declared and are available as variables (unless => "/bin/test -d ${path}") within the definition. Multiple variables (separated by commas) can be specified. Default values can also be specified for any parameter with =, and any parameter which has a default becomes non-mandatory when a resource of the new type is declared.

Defined types have a number of built-in variables available, including $name and $title, which are set to the title of the resource when it is declared. (The reasons for having two identical variables with this information are outside the scope of this document, and these two special variables cannot be used the same way in classes or other resources.) As of Puppet 2.6.5, the $name and $title variables can also be used as default values for parameters:

```
define svn_repo($path = "/var/${name}") {...}
```

Any metaparameters used when a defined resource is declared are also made available in the definition as variables:

```
define svn_repo($path) {
  exec { "create_repo_${name}":
    command => "/usr/bin/svnadmin create ${path}/${title}",
    unless  => "/bin/test -d ${path}",
  }
  if $require {
    Exec["create_repo_${name}"] {
      require +> $require,
    }
  }
}

svn_repo { 'puppet':
  path    => '/var/svn',
  require => Package['subversion'],
}
```

The above is perhaps not a perfect example, as most likely we would know that subversion was always required for svn checkouts, but it illustrates how require and other metaparameters can be used in defined types.

Defined resource types can have namespace separators (::) in their names, just like classes. When making a resource reference (e.g. File['/etc/motd']) to an instance of a defined type, you must capitalize all segments of the type's name (e.g. Apache::Vhost['wordpress']).

Classes vs. Defined Resource Types

Classes and defined types are created similarly, but they are used very differently.

Defined types are used to define reusable objects which will have multiple instances on a given host, so they cannot include any resources that will only have one instance. For instance, multiple uses of the same define cannot create the same file.

Classes, on the other hand, are guaranteed to be singletons — you can include them as many times as you want and you'll only ever get one copy of the resources.

Most often, services will be defined in a class, where the service's package, configuration files, and running service will all be gathered, because there will normally be one copy of each on a given host. (This idiom is sometimes referred to as "service-package-file").

Defined types would be used to manage resources like virtual hosts, of which you can have many, or to encode some simple information in a reusable wrapper to save typing.

Modules

You can (and should!) combine collections of classes, defined types, and resources into modules. Modules are portable collections of configuration, for example a module might contain all the resources required to configure Postfix or Apache. You can find out more on the Modules Page

Chaining resources

As of puppet version 2.6.0, resources may be chained together to declare relationships between and among them.

You can now specify relationships directly as statements in addition to the before and require resource metaparameters of previous versions:

```
File['/etc/ntp.conf'] -> Service['ntpd']
```

Manage the ntp configuration file before the ntpd service

You can specify a "notify" relationship by employing the tilde instead of the hyphen:

```
File['/etc/ntp.conf'] ~> Service['ntpd']
```

This manages the ntp configuration file before the ntpd service and notifies the service of changes to the ntp configuration file.

You can also do relationship chaining, specifying multiple relationships on a single line:

```
Package['ntp'] -> File['/etc/ntp.conf'] -> Service['ntpd']
```

Here we first manage the ntp package, second manage the ntp configuration file, and third manage the ntpd service.

Note that while it's confusing, you don't have to have all of the arrows be the same direction:

```
File['/etc/ntp.conf'] -> Service['ntpd'] <- Package['ntp']
```

Here the ntpd service requires /etc/ntp.conf and the ntp package.

Please note, relationships declared in this manner are between adjacent resources. In this example, the ntp package and the ntp configuration file are related to each other and puppet may try to manage the configuration file before the package is even installed, which may not be the desired behavior.

Chaining in this manner can provide some succinctness at the cost of readability.

You can also specify relationships when resources are declared, in addition to the above resource reference examples:

```
package { 'ntp': } -> file { '/etc/ntp.conf': }
```

Here we manage the ntp package before the ntp configuration file.

But wait! There's more! You can also specify a collection on either side of the relationship marker:

```
yumrepo { 'localyumrepo': .... }
package { 'ntp': provider => yum, ... }
Yumrepo <| |> -> Package <| provider == yum |>
```

This manages all yum repository resources before managing all package resources using the yum provider.

This, finally, provides easy many to many relationships in Puppet, but it also opens the door to massive dependency cycles. This last feature is a very powerful stick, and you can considerably hurt yourself with it. In particular, watch out when using virtual resources, as the collection operator realizes resources as a side-effect.

Nodes

Having knowledge of resources, classes, defines, and modules gets you to understanding of most of Puppet. Nodes are a very simple remaining step, which are how we map the what we define ("this is what a webserver looks like") to what machines are chosen to fulfill those instructions.

Node definitions look just like classes, including supporting inheritance, but they are special in that when a node (a managed computer running the Puppet client) connects to the Puppet master daemon, its name will be looked for in the list of defined nodes. The information found for the node will then be evaluated for that node, and then node will be sent that configuration.

Node names can be the short host name, or the fully qualified domain name (FQDN). Some names, especially fully qualified ones, need to be quoted, so it is a best practice to quote all of them. Here's an example:

```
node 'www.testing.com' {
  include common
  include apache, squid
}
```

The previous node definition creates a node called www.testing.com and includes the common, apache and squid classes.

You can also specify that multiple nodes receive an identical configuration by separating each with a comma:

```
node 'www.testing.com', 'www2.testing.com', 'www3.testing.com' {
  include common
  include apache, squid
}
```

The previous examples creates three identical nodes: www.testing.com, www2.testing.com, and www3.testing.com.

Matching Nodes with Regular Expressions

In Puppet 0.25.0 and later, nodes can also be matched by regular expressions, which is much more convenient than listing them individually, one-by-one:

```
node /^www\d+$/ {
  include common
}
```

The above would match any host called www and ending with one or more digits. Here's another example:

```
node /^(foo|bar)\.testing\.com$/ {
  include common
}
```

The above example would match either host foo or bar in the testing.com domain.

What happens if there are multiple regular expressions or node definitions set in the same file?

- If there is a node without a regular expression that matches the current client connecting, that will be used first.
- Otherwise the first matching regular expression wins.

Node Inheritance

Nodes support a limited inheritance model. Like classes, nodes can only inherit from one other node:

```
node 'www2.testing.com' inherits 'www.testing.com' {
  include loadbalancer
}
```

In this node definition the www2.testing.com inherits any configuration specified for the www.testing.com node in addition to including the loadbalancer class. In other words, it does everything "www.testing.com" does, but also takes on some additional functionality.

Default Nodes

If you create a node named `default`, the node configuration for default will be used if no other node matches are found.

External Nodes

In some cases you may already have an external list of machines and what roles they perform. This may be in LDAP, version control, or a database. You may also need to pass some variables to those nodes (more on variables later).

In these cases, writing an External Nodes script can help, and that can take the place of your node definitions. See that section for more information.

Additional Language Features

We've already gone over features such as ordering and grouping, though there's still a few more things to learn.

Puppet is not a programming language, it is a way of describing your IT infrastructure as a model. This is usually quite sufficient to get the job done, and prevents you from having to write a lot of programming code.

Quoting

Most of the time, you don't have to quote strings in Puppet. Any alphanumeric string starting with a letter (hyphens are also allowed), can leave out the quotes, though it's a best practice to quote strings for any non-native value.

Variable Interpolation With Quotes

So far, we've mentioned variables in terms of defines. If you need to use those variables within a string, use double quotes, not single quotes. Single-quoted strings will not do any variable interpolation, double-quoted strings will. Variables in strings can also be bracketed with {} which makes them easier to use together, and also a bit cleaner to read:

```
$value = "${one}${two}"
```

To put a quote character or `$` in a double-quoted string where it would normally have a special meaning, precede it with an escaping \. For an actual \, use \\.

We recommend using single quotes for all strings that do not require variable interpolation. Use double quotes for those strings that require variable interpolation. The Style Guide also discusses this with examples.

Capitalization

Capitalization of resources is used in three major ways:

- Referencing: when you want to reference an already declared resource, usually for dependency purposes, you have to capitalize the name of the resource, for example

```
require => File['sshdconfig']
```

- Inheritance. When overwriting the resource settings of a parent class from a subclass, use the uppercase versions of the resource names. Using the lowercase versions will result in an error. See the inheritance section above for an example of this.
- Setting default attribute values: Resource Defaults. As mentioned previously, using a capitalized resource with no `title` works to set the defaults for that resource. Our previous example was setting the default path for command executions.

Note that when capitalizing a namespaced defined type, you have to capitalize all segments of the type's name, e.g. `Apache::Vhost['wordpress']`.

Arrays

As mentioned in the class and resource examples above, Puppet allows usage of arrays in various areas. Arrays defined in puppet look like this:

```
[ 'one', 'two', 'three' ]
```

You can access array entries by their index, for example:

```
$foo = [ 'one', 'two', 'three' ]
notice $foo[1]
```

Would return two.

Several type members, such as 'alias' in the 'host' definition accept arrays as their value. A host resource with multiple aliases would look like this:

```
host { 'one.example.com':
  ensure => present,
  alias  => [ 'satu', 'dua', 'tiga' ],
  ip     => '192.168.100.1',
}
```

This would add a host 'one.example.com' to the hosts list with the three aliases 'satu', 'dua', and 'tiga'.

Or, for example, if you want a resource to require multiple other resources, the way to do this would be like this:

```
resource { 'baz':
  require  => [ Package['foo'], File['bar'] ],
}
```

Another example for array usage is to call a custom defined resource multiple times, like this:

```
define php::pear() {
  package { "php-${name}": ensure => installed }
}

php::pear { ['ldap', 'mysql', 'ps', 'snmp', 'sqlite', 'tidy', 'xmlrpc']: }
```

Of course, this can be used for native types as well:

```
file { [ 'foo', 'bar', 'foobar' ]:
  owner => 'root',
  group => 'root',
  mode  => '0600',
}
```

Hashes

Since Puppet version 2.6.0, hashes have been supported in the language. These hashes are defined like Ruby hashes using the form:

```
{ key1 => val1, key2 => val2, ... }
```

The hash keys are strings, but hash values can be any possible RHS values allowed in the language like function calls, variables, etc.

It is possible to assign hashes to a variable like so:

```
$myhash = { key1 => 'myval', key2 => $b }
```

And to access hash members (recursively) from a variable containing a hash (this also works for arrays too):

```
$myhash = { key => { subkey => 'b' }}
notice($myhash[key][subkey])
```

You can also use a hash member as a resource title, as a default definition parameter, or potentially as the value of a resource parameter,

Variables

Puppet supports variables like most other languages you may be familiar with. Puppet variables are denoted with $:

```
$content = 'some content\n'

file { '/tmp/testing': content => $content }
```

Puppet language is a declarative language, which means that its scoping and assignment rules are somewhat different than a normal imperative language. The primary difference is that you cannot change the value of a variable within a single scope, because that would rely on order in the file to determine the value of the variable. Order does not matter in a declarative language. Doing so will result in an error:

```
$user = root
file { '/etc/passwd':
  owner => $user,
}
$user = bin
file { '/bin':
  owner   => $user,
  recurse => true,
}
```

Rather than reassigning variables, instead use the built in conditionals:

```
$group = $operatingsystem ? {
  solaris => 'sysadmin',
  default => 'wheel',
}
```

A variable may only be assigned once per scope. However you still can set the same variable in non-overlapping scopes. For example, to set top-level configuration values:

```
node a {
  $setting = 'this'
  include class_using_setting
}
node b {
  $setting = 'that'
  include class_using_setting
}
```

In the above example, nodes “a” and “b” have different scopes, so this is not reassignment of the same variable.

Variable Scope

Scoping may initially seem like a foreign concept, though in reality it is pretty simple. A scope defines where a variable is valid. Unlike early programming languages like BASIC, variables are only valid and accessible in certain places in a program. Using the same variable name in different parts of the language do not refer to the same value.

Classes and nodes introduce new scopes. Puppet is currently dynamically scoped, which means that scope hierarchies are created based on where the code is evaluated instead of where the code is defined.

For example:

```
$test = 'top'
class myclass {
  exec { "/bin/echo ${test}": logoutput => true }
}

class other {
  $test = 'other'
  include myclass
}

include other
```

In this case, there's a top-level scope, a new scope for other, and the a scope below that for myclass. When this code is evaluated, $test evaluates to other, not top.

Qualified Variables

Puppet supports qualification of variables inside a class. This allows you to use variables defined in other classes.

For example:

```
class myclass {
  $test = 'content'
}

class anotherclass {
  $other = $myclass::test
}
```

In this example, the value of the $other variable evaluates to content. Qualified variables are read-only — you cannot set a variable's value from other class.

Variable qualification is dependent on the evaluation order of your classes. Class myclass must be evaluated before class anotherclass for variables to be set correctly.

Facts as Variables

In addition to user-defined variables, the facts generated by Facter are also available as variables. This allows values that you would see by running facter on a client system within Puppet manifests and also within Puppet templates. To use a fact as a variable prefix the name of the fact with $. For example, the value of the operatingsystem and puppetversion facts would be available as the variables $operatingsystem and $puppetversion.

Variable Expressions

In Puppet 0.24.6 and later, arbitrary expressions can be assigned to variables, for example:

```
$inch_to_cm = 2.54
$rack_length_cm = 19 * $inch_to_cm
$gigabyte = 1024 * 1024 * 1024
$can_update = ($ram_gb * $gigabyte) > 1 << 24
```

See the Expression section later on this page for further details of the expressions that are now available.

Appending to Variables

In Puppet 0.24.6 and later, values can be appended to array variables:

```
$ssh_users = [ 'myself', 'someone' ]

class test {
  $ssh_users += ['someone_else']
}
```

Here the $ssh_users variable contains an array with the elements myself and someone. Using the variable append syntax, +=, we added another element, someone_else to the array.

Please note, variables cannot be modified in the same scope because of the declarative nature of Puppet. As a result, $ssh_users contains the element 'someone_else' only in the scope of class test and not outside scopes. Resources outside of this scope will "see" the original array containing only myself and someone.

Conditionals

At some point you'll need to use a different value based on the value of a variable, or decide to not do something if a particular value is set.

Puppet currently supports two types of conditionals:

- The *selector* which can be used within resources and variable assignments to pick the correct value for an attribute, and

- *statement conditionals* which can be used more widely in your manifests to include additional classes, define distinct sets of resources within a class, or make other structural decisions.

Case statements do not return a value. Selectors do. That is the primary difference between them and why you would use one and not the other.

Selectors

If you're familiar with programming terms, The selector syntax works like a multi-valued ternary operator, similar to C's `foo = bar ? 1 : 0` operator where `foo` will be set to `1` if `bar` evaluates to true and `0` if `bar` is false.

Selectors are useful to specify a resource attribute or assign a variable based on a fact or another variable. In addition to any number of specified values, selectors also allow you to specify a default if no value matches; if no default is supplied and a selector fails to match, it will result in a parse error.

Here's a simple example of selector use:

```
file { '/etc/config':
  owner => $operatingsystem ? {
    'sunos'  => 'adm',
    'redhat' => 'bin',
    default  => undef,
  },
}
```

If the `$operatingsystem` fact (sent up from 'facter') returns `sunos` or `redhat` then the ownership of the file is set to `adm` or `bin` respectively. Any other result and the `owner` attribute will not be set, because it is listed as `undef`.

Remember to quote the comparators you're using in the selector as the lack of quotes can cause syntax errors.

Selectors can also be used in variable assignment:

```
$owner = $operatingsystem ? {
  'sunos'  => 'adm',
  'redhat' => 'bin',
  default  => undef,
}
```

In Puppet 0.25.0 and later, selectors can also be used with regular expressions:

```
$owner = $operatingsystem ? {
  /(redhat|debian)/ => 'bin',
  default           => undef,
}
```

In this last example, if `$operatingsystem` value matches either redhat or debian, then `bin` will be the selected result, otherwise the owner will not be set (`undef`).

Like Perl and some other languages with regular expression support, captures in selector regular expressions automatically create some limited scope variables (`$0` to `$n`):

```
$system = $operatingsystem ? {
  /(redhat|debian)/ => "our system is $1",
  default           => "our system is unknown",
}
```

In this last example, `$1` will get replaced by the content of the capture (here either `redhat` or `debian`).

The variable `$0` will contain the whole match.

Case Statement

`Case` is the other form of Puppet's two conditional statements, which can be wrapped around any Puppet code to add decision-making logic to your manifests. Case statements, unlike selectors, do not return a value. Also unlike selectors, a failed match without a default specified will simply skip the case statement instead of throwing a parse error. A common use for the `case` statement is to apply different classes to a particular node based on its operating system:

```
case $operatingsystem {
  'sunos':  { include solaris } # apply the solaris class
  'redhat': { include redhat  } # apply the redhat class
  default:  { include generic } # apply the generic class
}
```

Case statements can also specify multiple match conditions by separating each with a comma:

```
case $hostname {
  'jack','jill':     { include hill    } # apply the hill class
  'humpty','dumpty': { include wall    } # apply the wall class
  default:           { include generic } # apply the generic class
}
```

Here, if the $hostname fact returns either jack or jill the hill class would be included.

In Puppet 0.25.0 and later, the case statement also supports regular expressions:

```
case $hostname {
  /^j(ack|ill)$/: { include hill    } # apply the hill class
  /^[hd]umpty$/:  { include wall    } # apply the wall class
  default:        { include generic } # apply the generic class
}
```

In this last example, if $hostname matches either jack or jill, then the hill class will be included. But if $hostname matches either humpty or dumpty, then the wall class will be included.

As with selectors (see above), regular expressions captures are also available. These create limited scope variables $0 to $n:

```
case $hostname {
  /^j(ack|ill)$/: { notice("Welcome $1!") }
  default:        { notice("Welcome stranger") }
}
```

In this last example, if $host is jack or jill then a notice message will be logged with $1 replaced by either ack or ill. $0 contains the whole match.

If/Else Statement

The if/else provides branching options based on the truth value of a variable:

```
if $variable {
  file { '/some/file': ensure => present }
} else {
  file { '/some/other/file': ensure => present }
}
```

In Puppet 0.24.6 and later, the if statement can also branch based on the value of an expression:

```
if $server == 'mongrel' {
  include mongrel
} else {
  include nginx
}
```

In the above example, if the value of the variable $server is equal to mongrel, Puppet will include the class mongrel, otherwise it will include the class nginx.

From version 2.6.0 and later an elsif construct was introduced into the language:

```
if $server == 'mongrel' {
  include mongrel
} elsif $server == 'nginx' {
  include nginx
} else {
  include thin
}
```

Arithmetic expressions are also possible, for example:

```
if $ram > 1024 {
  $maxclient = 500
}
```

In the previous example if the value of the variable $ram is greater than 1024, Puppet will set the value of the $maxclient variable to 500.

"If" statements also support the use of regular expressions and "in" expressions. More complex expressions can also be made by combining arbitrary expressions with the Boolean and, or, and not operators:

```
if ( $processor_count > 2 ) and (( $ram >= 16 * $gigabyte ) or ( $disksize > 1000 )) {
  include for_big_irons
} else {
  include for_small_box
}
```

See the Expressions section further down for more information on expressions.

Virtual Resources

See Virtual Resources.

Virtual resources are available in Puppet 0.20.0 and later.

Virtual resources are resources that are not sent to the client unless realized.

The syntax for a virtual resource is:

```
@user { 'luke': ensure => present }
```

The user luke is now defined virtually. To realize that definition, you can use a collection:

```
User <| title == luke |>
```

This can be read as 'the user whose title is luke'. This is equivalent to using the realize function:

```
realize User['luke']
```

Realization could also use other criteria, such as realizing Users that match a certain group, or using a metaparameter like 'tag'.

The motivation for this feature is somewhat complicated; please see the Virtual Resources page for more information.

Exported Resources

Exported resources are an extension of virtual resources used to allow different hosts managed by Puppet to influence each other's Puppet configuration. This is described in detail on the Exported Resources page. As with virtual resources, new syntax was added to the language for this purpose.

The key syntactical difference between virtual and exported resources is that the special sigils (@ and <| |>) are doubled (@@ and «| |>>) when referring to an exported resource.

Here is an example with exported resources that shares SSH keys between clients:

```
class ssh {
  @@sshkey { $hostname: type => dsa, key => $sshdsakey }
  Sshkey <<| |>>
}
```

In the above example, notice that fulfillment and exporting are used together, so that any node that gets the 'sshkey' class will have all the ssh keys of other hosts. This could be done differently so that the keys could be realized on different hosts.

To actually work, the storeconfig parameter must be set to true in puppet.conf. This allows configurations from client to be stored on the central server.

The details of this feature are somewhat complicated; see the Exported Resources page for more information.

Reserved Words & Acceptable Characters

Variable names can include alphanumeric characters and underscores, and are case-sensitive.

Class names, module names, and the names of defined and custom resource types should be restricted to lowercase alphanumeric characters and underscores, and should begin with a lowercase letter; that is, they should match the expression `[a-z][a-z0-9_]*`. Although some names that violate these restrictions currently work, using them is not recommended.

Class and defined resource type names can use `::` as a namespace separator, which is both semantically useful and a means of directing the behavior of the module autoloader. The final segment of a qualified variable name must obey the restrictions on variable names, and the preceding segments must obey the restrictions on class names.

Parameters used in parameterized classes and defined resource types can include alphanumeric characters and underscores, cannot begin with an underscore, and are case-sensitive. In practice, they should be treated as though they were under the same restrictions as class names in order to maximize future compatibility.

There is no practical restriction on resource names.

Any word that the syntax uses for special meaning is a reserved word, meaning you cannot use it for variable or type names. Words like `true`, `define`, `inherits`, and `class` are all reserved. If you ever need to use a reserved word as a value, be sure to quote it.

Comments

Puppet supports two types of comments:

- Unix shell style comments; they can either be on their own line or at the end of a line.
- multi-line C-style comments (available in Puppet 0.24.7 and later)

Here is a shell style comment:

```
# this is a comment
```

You can see an example of a multi-line comment:

```
/*
this is a comment
*/
```

Expressions

Starting with version 0.24.6 the Puppet language supports arbitrary expressions in `if` statement boolean tests and in the right hand value of variable assignments.

Puppet expressions can be composed of:

- boolean expressions, which are combination of other expressions combined by boolean operators (`and`, `or` and `not`)
- comparison expressions, which consist of variables, numerical operands or other expressions combined with comparison operators (==, !=, <, >, <=, >, >=)
- arithmetic expressions, which consists of variables, numerical operands or other expressions combined with the following arithmetic operators: +, −, /, *, <<, >>
- in Puppet 0.25.0 and later, regular expression matches with the help of the regex match operator: `=~` and `!~`
- in Puppet 2.6.0 and later, `in` expressions, which test whether the right operand contains the left operand.

Expressions can be enclosed in parenthesis, `()`, to group expressions and resolve operator ambiguity.

Operator precedence

The Puppet operator precedence conforms to the standard precedence in most systems, from highest to lowest:

```
! -> not
* / -> times and divide
- + -> minus, plus
<< >> -> left shift and right shift
== != -> not equal, equal
>= <= > < -> greater equal, less or equal, greater than, less than
and
or
```

Expression examples

Comparison expressions

Comparison expressions include tests for equality using the == expression:

```
if $variable == 'foo' {
  include bar
} else {
  include foobar
}
```

Here if $variable has a value of foo, Puppet will then include the bar class, otherwise it will include the foobar class.

Here is another example shows the use of the != ('not equal') comparison operator:

```
if $variable != 'foo' {
  $othervariable = 'bar'
} else {
  $othervariable = 'foobar'
}
```

In our second example if $variable is not equal to a value of foo, Puppet will then set the value of the $othervariable variable to bar, otherwise it will set the $othervariable variable to foobar.

Note that comparison of strings is case-insensitive.

Arithmetic expressions

You can also perform a variety of arithmetic expressions, for example:

```
$one = 1
$one_thirty = 1.30
$two = 2.034e-2

$result = ((( $two + 2) / $one_thirty) + 4 * 5.45) - (6 << ($two + 4)) + (0x800 + -9)
```

Boolean expressions

Boolean expressions are also possible using or, and and not:

```
$one = 1
$two = 2
$var = ( $one < $two ) and ( $one + 1 == $two )
```

The exclamation mark (!) can be used as a synonym for not.

Regular expressions

In Puppet 0.25.0 and later, Puppet supports regular expression matching using =~ (match) and !~ (not-match) for example:

```
if $host =~ /^www(\d+)\./ {
  notice('Welcome web server #$1')
}
```

Like case and selectors, the regex match operators create limited scope variables for each regex capture. In the previous example, $1 will be replaced by the number following www in $host. Those variables are valid only for the statements inside the braces of the if clause.

"in" expressions

From Puppet 2.6.0, Puppet supports an "in" syntax. This operator allows you to find if the left operand is in the right one. The left operand must be a string, but the right operand can be:

- a string
- an array
- a hash (the search is done on the keys)

This syntax can be used in any place where an expression is supported:

```
$eatme = 'eat'
if $eatme in ['ate', 'eat'] {
...
}

$value = 'beat generation'
if 'eat' in $value {
  notice('on the road')
}
```

Like other expressions, "in" expressions can be combined or negated with boolean operators:

```
if ! ($eatme in ['ate', 'eat']) { ... }
```

Backus Naur Form

We've already covered the list of operators, though if you wish to see it, here's the available operators in Backus Naur Form:

```
<exp> ::=  <exp> <arithop> <exp>
         | <exp> <boolop> <exp>
         | <exp> <compop> <exp>
         | <exp> <matchop> <regex>
         | ! <exp>
         | - <exp>
         | "(" <exp> ")"
         | <rightvalue>

<arithop> ::= "+" | "-" | "/" | "*" | "<<" | ">>"
<boolop>  ::= "and" | "or"
<compop>  ::= "==" | "!=" | ">" | ">=" | "<=" | "<"
<matchop>  ::= "=~" | "!~"

<rightvalue> ::= <variable> | <function-call> | <literals>
<literals> ::= <float> | <integer> | <hex-integer> | <octal-integer> | <quoted-string>
<regex> ::= '/regex/'
```

Functions

Puppet supports many built in functions; see the Function Reference for details — see Custom Functions for information on how to create your own custom functions.

Some functions can be used as a statement:

```
notice('Something weird is going on')
```

(The notice function above is an example of a function that will log on the server)

Or without parentheses:

```
notice 'Something weird is going on'
```

Some functions instead return a value:

```
file { '/my/file': content => template('mytemplate.erb') }
```

All functions run on the Puppet master, so you only have access to the file system and resources on that host from your functions. The only exception to this is that the value of any Facter facts that have been sent to the master from your clients are also at your disposal. See the Tools Guide for more information about these components.

Importing Manifests

Puppet has an import keyword for importing other manifests. Code in those external manifests should always be stored in a class or defined resource type, or else it will be imported into the main scope and applied to all nodes. Currently files are only searched for within the same directory as the file doing the importing.

Files can also be imported using globbing, as implemented by Ruby's Dir.glob method:

```
import 'classes/*.pp'
import 'packages/[a-z]*.pp'
```

Best practices calls for organizing manifests into Modules

Handling Compilation Errors

Puppet does not use manifests directly, it compiles them down to a internal format that the clients can understand.

By default, when a manifest fails to compile, the previously compiled version of the Puppet manifest is used instead.

This behavior is governed by a setting in puppet.conf called usecacheonfailure and is set by default to true.

This may result in surprising behaviour if you are editing complex configurations.

Running the Puppet client with --no-usecacheonfailure or with --test, or setting usecacheonfailure = false in the configuration file, will disable this behaviour.

Chapter 43

Puppet Application Manpages

View documentation for each of the Puppet executables.

- puppet agent
- puppet apply
- puppet cert
- puppet describe
- puppet device
- puppet doc
- puppet filebucket
- puppet inspect
- puppet kick
- puppet master
- puppet queue
- puppet resource

Chapter 44

puppet agent Manual Page

NAME

puppet−agent - The puppet agent daemon

SYNOPSIS

Retrieves the client configuration from the puppet master and applies it to the local host.

This service may be run as a daemon, run periodically using cron (or something similar), or run interactively for testing purposes.

USAGE

puppet agent [-D|—daemonize|—no-daemonize] [-d|—debug] [—detailed-exitcodes] [—disable] [—enable] [-h|—help] [—certname host name] [-l|—logdest syslog|file|console] [-o|—onetime] [—serve handler] [-t|—test] [—noop] [—digest digest] [—fingerprint] [-V|—version] [-v|—verbose] [-w|—waitforcert seconds]

DESCRIPTION

This is the main puppet client. Its job is to retrieve the local machine's configuration from a remote server and apply it. In order to successfully communicate with the remote server, the client must have a certificate signed by a certificate authority that the server trusts; the recommended method for this, at the moment, is to run a certificate authority as part of the puppet server (which is the default). The client will connect and request a signed certificate, and will continue connecting until it receives one.

Once the client has a signed certificate, it will retrieve its configuration and apply it.

USAGE NOTES

'puppet agent' does its best to find a compromise between interactive use and daemon use. Run with no arguments and no configuration, it will go into the background, attempt to get a signed certificate, and retrieve and apply its configuration every 30 minutes.

Some flags are meant specifically for interactive use — in particular, 'test', 'tags' or 'fingerprint' are useful. 'test' enables verbose logging, causes the daemon to stay in the foreground, exits if the server's configuration is invalid (this happens if, for instance, you've left a syntax error on the server), and exits after running the configuration once (rather than hanging around as a long-running process).

'tags' allows you to specify what portions of a configuration you want to apply. Puppet elements are tagged with all of the class or definition names that contain them, and you can use the 'tags' flag to specify one of these names, causing only configuration elements contained within that class or definition to be applied. This is very useful when you are testing new configurations — for instance, if you are just starting to manage 'ntpd',

you would put all of the new elements into an 'ntpd' class, and call puppet with '—tags ntpd', which would only apply that small portion of the configuration during your testing, rather than applying the whole thing.

'fingerprint' is a one-time flag. In this mode 'puppet agent' will run once and display on the console (and in the log) the current certificate (or certificate request) fingerprint. Providing the '—digest' option allows to use a different digest algorithm to generate the fingerprint. The main use is to verify that before signing a certificate request on the master, the certificate request the master received is the same as the one the client sent (to prevent against man-in-the-middle attacks when signing certificates).

OPTIONS

Note that any configuration parameter that's valid in the configuration file is also a valid long argument. For example, 'server' is a valid configuration parameter, so you can specify '—server servername' as an argument.

See the configuration file documentation at http://docs.puppetlabs.com/references/stable/configuration.html for the full list of acceptable parameters. A commented list of all configuration options can also be generated by running puppet agent with '—genconfig'.

—**daemonize** Send the process into the background. This is the default.

—**no-daemonize** Do not send the process into the background.

—**debug** Enable full debugging.

—**digest** Change the certificate fingerprinting digest algorithm. The default is MD5. Valid values depends on the version of OpenSSL installed, but should always at least contain MD5, MD2, SHA1 and SHA256.

—**detailed-exitcodes** Provide transaction information via exit codes. If this is enabled, an exit code of '2' means there were changes, and an exit code of '4' means that there were failures during the transaction. This option only makes sense in conjunction with —onetime.

—**disable** Disable working on the local system. This puts a lock file in place, causing 'puppet agent' not to work on the system until the lock file is removed. This is useful if you are testing a configuration and do not want the central configuration to override the local state until everything is tested and committed.

'puppet agent' uses the same lock file while it is running, so no more than one 'puppet agent' process is working at a time.

'puppet agent' exits after executing this.

—**enable** Enable working on the local system. This removes any lock file, causing 'puppet agent' to start managing the local system again (although it will continue to use its normal scheduling, so it might not start for another half hour).

'puppet agent' exits after executing this.

—**certname** Set the certname (unique ID) of the client. The master reads this unique identifying string, which is usually set to the node's fully-qualified domain name, to determine which configurations the node will receive. Use this option to debug setup problems or implement unusual node identification schemes.

—**help** Print this help message

—**logdest** Where to send messages. Choose between syslog, the console, and a log file. Defaults to sending messages to syslog, or the console if debugging or verbosity is enabled.

—**no-client** Do not create a config client. This will cause the daemon to run without ever checking for its configuration automatically, and only makes sense

—**onetime** Run the configuration once. Runs a single (normally daemonized) Puppet run. Useful for interactively running puppet agent when used in conjunction with the —no-daemonize option.

—**fingerprint** Display the current certificate or certificate signing request fingerprint and then exit. Use the '—digest' option to change the digest algorithm used.

—**serve** Start another type of server. By default, 'puppet agent' will start a service handler that allows authenticated and authorized remote nodes to trigger the configuration to be pulled down and applied. You can specify any handler here that does not require configuration, e.g., filebucket, ca, or resource. The handlers are in 'lib/puppet/network/handler', and the names must match exactly, both in the call to 'serve' and in 'namespaceauth.conf'.

—**test** Enable the most common options used for testing. These are 'onetime', 'verbose', 'ignorecache', 'no-daemonize', 'no-usecacheonfailure', 'detailed-exit-codes', 'no-splay', and 'show_diff'.

—**noop** Use 'noop' mode where the daemon runs in a no-op or dry-run mode. This is useful for seeing what changes Puppet will make without actually executing the changes.

—**verbose** Turn on verbose reporting.

—**version** Print the puppet version number and exit.

—**waitforcert** This option only matters for daemons that do not yet have certificates and it is enabled by default, with a value of 120 (seconds). This causes 'puppet agent' to connect to the server every 2 minutes and ask it to sign a certificate request. This is useful for the initial setup of a puppet client. You can turn off waiting for certificates by specifying a time of 0.

EXAMPLE

```
$ puppet agent —server puppet.domain.com
```

AUTHOR

Luke Kanies

COPYRIGHT

Chapter 45

puppet apply Manual Page

NAME

puppet−apply - Apply Puppet manifests locally

SYNOPSIS

Applies a standalone Puppet manifest to the local system.

USAGE

puppet apply [-h|—help] [-V|—version] [-d|—debug] [-v|—verbose] [-e|—execute] [—detailed-exitcodes] [-l|—logdest file] [—apply catalog] file

DESCRIPTION

This is the standalone puppet execution tool; use it to apply individual manifests.

When provided with a modulepath, via command line or config file, puppet apply can effectively mimic the catalog that would be served by puppet master with access to the same modules, although there are some subtle differences. When combined with scheduling and an automated system for pushing manifests, this can be used to implement a serverless Puppet site.

Most users should use 'puppet agent' and 'puppet master' for site-wide manifests.

OPTIONS

Note that any configuration parameter that's valid in the configuration file is also a valid long argument. For example, 'modulepath' is a valid configuration parameter, so you can specify '—tags class,tag' as an argument.

See the configuration file documentation at http://docs.puppetlabs.com/references/stable/configuration.html for the full list of acceptable parameters. A commented list of all configuration options can also be generated by running puppet with '—genconfig'.

—**debug** Enable full debugging.

—**detailed-exitcodes** Provide transaction information via exit codes. If this is enabled, an exit code of '2' means there were changes, and an exit code of '4' means that there were failures during the transaction.

—**help** Print this help message

—**loadclasses** Load any stored classes. 'puppet agent' caches configured classes (usually at /etc/puppet/-classes.txt), and setting this option causes all of those classes to be set in your puppet manifest.

—**logdest** Where to send messages. Choose between syslog, the console, and a log file. Defaults to sending messages to the console.

—**execute** Execute a specific piece of Puppet code

—**verbose** Print extra information.

—**apply** Apply a JSON catalog (such as one generated with 'puppet master —compile'). You can either specify a JSON file or pipe in JSON from standard input.

EXAMPLE

```
$ puppet apply -l /tmp/manifest.log manifest.pp
$ puppet apply --modulepath=/root/dev/modules -e "include ntpd::server"
```

AUTHOR

Luke Kanies

COPYRIGHT

Chapter 46

puppet cert Manual Page

NAME

puppet−cert - Manage certificates and requests

SYNOPSIS

Standalone certificate authority. Capable of generating certificates, but mostly used for signing certificate requests from puppet clients.

USAGE

puppet cert action [-h|—help] [-V|—version] [-d|—debug] [-v|—verbose] [—digest digest] [host]

DESCRIPTION

Because the puppet master service defaults to not signing client certificate requests, this script is available for signing outstanding requests. It can be used to list outstanding requests and then either sign them individually or sign all of them.

ACTIONS

Every action except 'list' and 'generate' requires a hostname to act on, unless the '—all' option is set.

clean Revoke a host's certificate (if applicable) and remove all files related to that host from puppet cert's storage. This is useful when rebuilding hosts, since new certificate signing requests will only be honored if puppet cert does not have a copy of a signed certificate for that host. If '—all' is specified then all host certificates, both signed and unsigned, will be removed.

fingerprint Print the DIGEST (defaults to md5) fingerprint of a host's certificate.

generate Generate a certificate for a named client. A certificate/keypair will be generated for each client named on the command line.

list List outstanding certificate requests. If '—all' is specified, signed certificates are also listed, prefixed by '+', and revoked or invalid certificates are prefixed by '-' (the verification outcome is printed in parenthesis).

print Print the full-text version of a host's certificate.

revoke Revoke the certificate of a client. The certificate can be specified either by its serial number (given as a decimal number or a hexadecimal number prefixed by '0x') or by its hostname. The certificate is revoked by adding it to the Certificate Revocation List given by the 'cacrl' configuration option. Note that the puppet master needs to be restarted after revoking certificates.

sign Sign an outstanding certificate request.

verify Verify the named certificate against the local CA certificate.

OPTIONS

Note that any configuration parameter that's valid in the configuration file is also a valid long argument. For example, 'ssldir' is a valid configuration parameter, so you can specify '—ssldir directory' as an argument.

See the configuration file documentation at http://docs.puppetlabs.com/references/stable/configuration.html for the full list of acceptable parameters. A commented list of all configuration options can also be generated by running puppet cert with '—genconfig'.

—**all** Operate on all items. Currently only makes sense with the 'sign', 'clean', 'list', and 'fingerprint' actions.

—**digest** Set the digest for fingerprinting (defaults to md5). Valid values depends on your openssl and openssl ruby extension version, but should contain at least md5, sha1, md2, sha256.

—**debug** Enable full debugging.

—**help** Print this help message

—**verbose** Enable verbosity.

—**version** Print the puppet version number and exit.

EXAMPLE

```
$ puppet cert list
culain.madstop.com
$ puppet cert sign culain.madstop.com
```

AUTHOR

Luke Kanies

COPYRIGHT

Chapter 47

puppet describe Manual Page

NAME

puppet−describe - Display help about resource types

SYNOPSIS

Prints help about Puppet resource types, providers, and metaparameters.

USAGE

puppet describe [-h|—help] [-s|—short] [-p|—providers] [-l|—list] [-m|—meta]

OPTIONS

—**help** Print this help text

—**providers** Describe providers in detail for each type

—**list** List all types

—**meta** List all metaparameters

—**short** List only parameters without detail

EXAMPLE

```
$ puppet describe —list
$ puppet describe file —providers
$ puppet describe user −s −m
```

AUTHOR

David Lutterkort

COPYRIGHT

Chapter 48

puppet device Manual Page

NAME

puppet−device - Manage remote network devices

SYNOPSIS

Retrieves all configurations from the puppet master and apply them to the remote devices configured in /etc/puppet/device.conf.

Currently must be run out periodically, using cron or something similar.

USAGE

puppet device [-d|—debug] [—detailed-exitcodes] [-V|—version]

```
[−h|−−help] [−l|−−logdest syslog|<file>|console]
[−v|−−verbose] [−w|−−waitforcert <seconds>]
```

DESCRIPTION

Once the client has a signed certificate for a given remote device, it will retrieve its configuration and apply it.

USAGE NOTES

One need a /etc/puppet/device.conf file with the following content:

[remote.device.fqdn] type type url url

where: * type: the current device type (the only value at this time is cisco) * url: an url allowing to connect to the device

Supported url must conforms to: scheme://user:password@hostname/?query

with: * scheme: either ssh or telnet * user: username, can be omitted depending on the switch/router configuration * password: the connection password * query: this is device specific. Cisco devices supports an enable parameter whose value would be the enable password.

OPTIONS

Note that any configuration parameter that's valid in the configuration file is also a valid long argument. For example, 'server' is a valid configuration parameter, so you can specify '—server servername' as an argument.

—**debug** Enable full debugging.

—**detailed-exitcodes** Provide transaction information via exit codes. If this is enabled, an exit code of '2' means there were changes, and an exit code of '4' means that there were failures during the transaction. This option only makes sense in conjunction with —onetime.

—**help** Print this help message

—**logdest** Where to send messages. Choose between syslog, the console, and a log file. Defaults to sending messages to syslog, or the console if debugging or verbosity is enabled.

—**verbose** Turn on verbose reporting.

—**waitforcert** This option only matters for daemons that do not yet have certificates and it is enabled by default, with a value of 120 (seconds). This causes +puppet agent+ to connect to the server every 2 minutes and ask it to sign a certificate request. This is useful for the initial setup of a puppet client. You can turn off waiting for certificates by specifying a time of 0.

EXAMPLE

```
$ puppet device —server puppet.domain.com
```

AUTHOR

Brice Figureau

COPYRIGHT

Chapter 49

puppet doc Manual Page

NAME

puppet−doc - Generate Puppet documentation and references

SYNOPSIS

Generates a reference for all Puppet types. Largely meant for internal Puppet Labs use.

USAGE

puppet doc [-a|—all] [-h|—help] [-o|—outputdir rdoc-outputdir] [-m|—mode text|pdf|rdoc] [-r|—reference reference-name] [—charset charset] [manifest-file]

DESCRIPTION

If mode is not 'rdoc', then this command generates a Markdown document describing all installed Puppet types or all allowable arguments to puppet executables. It is largely meant for internal use and is used to generate the reference document available on the Puppet Labs web site.

In 'rdoc' mode, this command generates an html RDoc hierarchy describing the manifests that are in 'manifest-dir' and 'modulepath' configuration directives. The generated documentation directory is doc by default but can be changed with the 'outputdir' option.

If the command is run with the name of a manifest file as an argument, puppet doc will output a single manifest's documentation on stdout.

OPTIONS

—**all** Output the docs for all of the reference types. In 'rdoc' modes, this also outputs documentation for all resources

—**help** Print this help message

—**outputdir** Specifies the directory where to output the rdoc documentation in 'rdoc' mode.

—**mode** Determine the output mode. Valid modes are 'text', 'pdf' and 'rdoc'. The 'pdf' mode creates PDF formatted files in the /tmp directory. The default mode is 'text'. In 'rdoc' mode you must provide 'manifests-path'

—**reference** Build a particular reference. Get a list of references by running 'puppet doc —list'.

—**charset** Used only in 'rdoc' mode. It sets the charset used in the html files produced.

EXAMPLE

```
$ puppet doc -r type > /tmp/type_reference.markdown
```

or

```
$ puppet doc --outputdir /tmp/rdoc --mode rdoc /path/to/manifests
```

or

```
$ puppet doc /etc/puppet/manifests/site.pp
```

or

```
$ puppet doc -m pdf -r configuration
```

AUTHOR

Luke Kanies

COPYRIGHT

Chapter 50

puppet filebucket Manual Page

NAME

puppet−filebucket - Store and retrieve files in a filebucket

SYNOPSIS

A stand-alone Puppet filebucket client.

USAGE

puppet filebucket mode [-h|—help] [-V|—version] [-d|—debug] [-v|—verbose] [-l|—local] [-r|—remote] [-s|—server server] [-b|—bucket directory] file file …

Puppet filebucket can operate in three modes, with only one mode per call:

backup: Send one or more files to the specified file bucket. Each sent file is printed with its resulting md5 sum.

get: Return the text associated with an md5 sum. The text is printed to stdout, and only one file can be retrieved at a time.

restore: Given a file path and an md5 sum, store the content associated with the sum into the specified file path. You can specify an entirely new path to this argument; you are not restricted to restoring the content to its original location.

DESCRIPTION

This is a stand-alone filebucket client for sending files to a local or central filebucket.

Note that 'filebucket' defaults to using a network-based filebucket available on the server named 'puppet'. To use this, you'll have to be running as a user with valid Puppet certificates. Alternatively, you can use your local file bucket by specifying '—local'.

OPTIONS

Note that any configuration parameter that's valid in the configuration file is also a valid long argument. For example, 'ssldir' is a valid configuration parameter, so you can specify '—ssldir directory' as an argument.

See the configuration file documentation at http://docs.puppetlabs.com/references/stable/configuration.html for the full list of acceptable parameters. A commented list of all configuration options can also be generated by running puppet with '—genconfig'.

—debug Enable full debugging.

—**help** Print this help message

—**local** Use the local filebucket. This will use the default configuration information.

—**remote** Use a remote filebucket. This will use the default configuration information.

—**server** The server to send the file to, instead of locally.

—**verbose** Print extra information.

—**version** Print version information.

EXAMPLE

```
$ puppet filebucket backup /etc/passwd
/etc/passwd: 429b225650b912a2ee067b0a4cf1e949
$ puppet filebucket restore /tmp/passwd 429b225650b912a2ee067b0a4cf1e949
```

AUTHOR

Luke Kanies

COPYRIGHT

Chapter 51

puppet inspect Manual Page

NAME

puppet−inspect - Send an inspection report

SYNOPSIS

Prepares and submits an inspection report to the puppet master.

USAGE

puppet inspect

DESCRIPTION

This command uses the cached catalog from the previous run of 'puppet agent' to determine which attributes of which resources have been marked as auditable with the 'audit' metaparameter. It then examines the current state of the system, writes the state of the specified resource attributes to a report, and submits the report to the puppet master.

Puppet inspect does not run as a daemon, and must be run manually or from cron.

OPTIONS

Any configuration setting which is valid in the configuration file is also a valid long argument, e.g. '—server=master.doma
See the configuration file documentation at http://docs.puppetlabs.com/references/latest/configuration.html for the full list of acceptable settings.

AUTHOR

Puppet Labs

COPYRIGHT

Chapter 52

puppet kick Manual Page

NAME

puppet−kick - Remotely control puppet agent

SYNOPSIS

Trigger a puppet agent run on a set of hosts.

USAGE

puppet kick [-a|—all] [-c|—class class] [-d|—debug] [-f|—foreground] [-h|—help] [—host host] [—no-fqdn] [—ignoreschedules] [-t|—tag tag] [—test] [-p|—ping] host [host [...]]

DESCRIPTION

This script can be used to connect to a set of machines running 'puppet agent' and trigger them to run their configurations. The most common usage would be to specify a class of hosts and a set of tags, and 'puppet kick' would look up in LDAP all of the hosts matching that class, then connect to each host and trigger a run of all of the objects with the specified tags.

If you are not storing your host configurations in LDAP, you can specify hosts manually.

You will most likely have to run 'puppet kick' as root to get access to the SSL certificates.

'puppet kick' reads 'puppet master"s configuration file, so that it can copy things like LDAP settings.

USAGE NOTES

Puppet kick is useless unless puppet agent is listening for incoming connections and allowing access to the run endpoint. This entails starting the agent with listen = true in its puppet.conf file, and allowing access to the /run path in its auth.conf file; see http://docs.puppetlabs.com/guides/rest_auth_conf.html for more details.

Additionally, due to a known bug, you must make sure a namespaceauth.conf file exists in puppet agent's $confdir. This file will not be consulted, and may be left empty.

OPTIONS

Note that any configuration parameter that's valid in the configuration file is also a valid long argument. For example, 'ssldir' is a valid configuration parameter, so you can specify '—ssldir directory' as an argument.

See the configuration file documentation at http://docs.puppetlabs.com/references/latest/configuration.html for the full list of acceptable parameters. A commented list of all configuration options can also be generated by running puppet master with '—genconfig'.

—**all** Connect to all available hosts. Requires LDAP support at this point.

—**class** Specify a class of machines to which to connect. This only works if you have LDAP configured, at the moment.

—**debug** Enable full debugging.

—**foreground** Run each configuration in the foreground; that is, when connecting to a host, do not return until the host has finished its run. The default is false.

—**help** Print this help message

—**host** A specific host to which to connect. This flag can be specified more than once.

—**ignoreschedules** Whether the client should ignore schedules when running its configuration. This can be used to force the client to perform work it would not normally perform so soon. The default is false.

—**parallel** How parallel to make the connections. Parallelization is provided by forking for each client to which to connect. The default is 1, meaning serial execution.

—**tag** Specify a tag for selecting the objects to apply. Does not work with the —test option.

—**test** Print the hosts you would connect to but do not actually connect. This option requires LDAP support at this point.

—**ping** Do a ICMP echo against the target host. Skip hosts that don't respond to ping.

EXAMPLE

```
$ sudo puppet kick -p 10 -t remotefile -t webserver host1 host2
```

AUTHOR

Luke Kanies

COPYRIGHT

Chapter 53

puppet master Manual Page

NAME

puppet−master - The puppet master daemon

SYNOPSIS

The central puppet server. Functions as a certificate authority by default.

USAGE

puppet master [-D|—daemonize|—no-daemonize] [-d|—debug] [-h|—help] [-l|—logdest file|console|syslog] [-v|—verbose] [-V|—version] [—compile node-name]

DESCRIPTION

This command starts an instance of puppet master, running as a daemon and using Ruby's built-in Webrick web-server. Puppet master can also be managed by other application servers; when this is the case, this executable is not used.

OPTIONS

Note that any configuration parameter that's valid in the configuration file is also a valid long argument. For example, 'ssldir' is a valid configuration parameter, so you can specify '—ssldir directory' as an argument.

See the configuration file documentation at http://docs.puppetlabs.com/references/stable/configuration.html for the full list of acceptable parameters. A commented list of all configuration options can also be generated by running puppet master with '—genconfig'.

—**daemonize** Send the process into the background. This is the default.

—**no-daemonize** Do not send the process into the background.

—**debug** Enable full debugging.

—**help** Print this help message.

—**logdest** Where to send messages. Choose between syslog, the console, and a log file. Defaults to sending messages to syslog, or the console if debugging or verbosity is enabled.

—**verbose** Enable verbosity.

—**version** Print the puppet version number and exit.

—**compile** Compile a catalogue and output it in JSON from the puppet master. Uses facts contained in the $vardir/yaml/ directory to compile the catalog.

EXAMPLE

puppet master

AUTHOR

Luke Kanies

COPYRIGHT

Chapter 54

puppet queue Manual Page

NAME

puppet−queue - Queuing daemon for asynchronous storeconfigs

SYNOPSIS

Retrieves serialized storeconfigs records from a queue and processes them in order.

USAGE

puppet queue [-d|—debug] [-v|—verbose]

DESCRIPTION

This application runs as a daemon and processes storeconfigs data, retrieving the data from a stomp server message queue and writing it to a database.

For more information, including instructions for properly setting up your puppet master and message queue, see the documentation on setting up asynchronous storeconfigs at: http://projects.puppetlabs.com/projects/1/wiki/Using_

OPTIONS

Note that any configuration parameter that's valid in the configuration file is also a valid long argument. For example, 'server' is a valid configuration parameter, so you can specify '—server servername' as an argument.

See the configuration file documentation at http://docs.puppetlabs.com/references/stable/configuration.html for the full list of acceptable parameters. A commented list of all configuration options can also be generated by running puppet queue with '—genconfig'.

—**debug** Enable full debugging.

—**help** Print this help message

—**verbose** Turn on verbose reporting.

—**version** Print the puppet version number and exit.

EXAMPLE

```
$ puppet queue
```

AUTHOR

Luke Kanies

COPYRIGHT

Chapter 55

puppet resource Manual Page

NAME

puppet−resource - The resource abstraction layer shell

SYNOPSIS

Uses the Puppet RAL to directly interact with the system.

USAGE

puppet resource [-h|—help] [-d|—debug] [-v|—verbose] [-e|—edit] [-H|—host host] [-p|—param parameter] [-t|—types] type [name] [attribute=value ...]

DESCRIPTION

This command provides simple facilities for converting current system state into Puppet code, along with some ability to modify the current state using Puppet's RAL.

By default, you must at least provide a type to list, in which case puppet resource will tell you everything it knows about all resources of that type. You can optionally specify an instance name, and puppet resource will only describe that single instance.

If given a type, a name, and a series of attribute=value pairs, puppet resource will modify the state of the specified resource. Alternately, if given a type, a name, and the '—edit' flag, puppet resource will write its output to a file, open that file in an editor, and then apply the saved file as a Puppet transaction.

OPTIONS

Note that any configuration parameter that's valid in the configuration file is also a valid long argument. For example, 'ssldir' is a valid configuration parameter, so you can specify '—ssldir directory' as an argument.

See the configuration file documentation at http://docs.puppetlabs.com/references/stable/configuration.html for the full list of acceptable parameters. A commented list of all configuration options can also be generated by running puppet with '—genconfig'.

—**debug** Enable full debugging.

—**edit** Write the results of the query to a file, open the file in an editor, and read the file back in as an executable Puppet manifest.

—**host** When specified, connect to the resource server on the named host and retrieve the list of resouces of the type specified.

—**help** Print this help message.

—**param** Add more parameters to be outputted from queries.

—**types** List all available types.

—**verbose** Print extra information.

EXAMPLE

This example uses puppet resource to return a Puppet configuration for the user luke:

```
$ puppet resource user luke
user { 'luke':
 home => '/home/luke',
 uid => '100',
 ensure => 'present',
 comment => 'Luke Kanies,,,',
 gid => '1000',
 shell => '/bin/bash',
 groups => ['sysadmin','audio','video','puppet']
}
```

AUTHOR

Luke Kanies

COPYRIGHT

Chapter 56

REST Access Control

Learn how to configure access to Puppet's REST API using the rest_authconfig file, a.k.a. auth.conf. **This document is currently being checked for accuracy. If you note any errors, please email them to faq@puppetlabs.com.**

REST

Puppet master and puppet agent communicate with each other over a RESTful network API. By default, the usage of this API is limited to the standard types of master/agent communications. However, it can be exposed to other processes and used to build advanced tools on top of Puppet's existing infrastructure and functionality. (REST API calls are formatted as https://{server}:{port}/{environment}/{resource}/{key}.)

As you might guess, this can be turned into a security hazard, so access to the REST API is strictly controlled by a special configuration file.

auth.conf

The official name of the file controlling REST API access, taken from the configuration option that sets its location, is rest_authconfig, but it's more frequently known by its default filename of auth.conf. If you don't set a different location for it, Puppet will look for the file at $confdir/auth.conf.

You cannot configure different environments to use multiple rest_authconfig files.

File Format

The auth.conf file consists of a series of ACLs (Access Control Lists); ACLs should be separated by double newlines. Lines starting with # are interpreted as comments.

```
# This is a comment
path /facts
method find, search
auth yes
allow custominventory.site.net, devworkstation.site.net

path /
auth any
allow devworkstation.site.net
```

Due to a known bug, trailing whitespace is not permitted after any line in auth.conf.

ACL format

Each auth.conf ACL is formatted as follows:

```
path [~] {/path/to/resource|regex}
[environment {list of environments}]
[method {list of methods}]
[auth[enthicated] {yes|no|on|off|any}]
[allow {hostname|certname|*}]
```

Lists of values are comma-separated, with an optional space after the comma.

Path

An ACL's path is interpreted as either a regular expression (with tilde) or a path prefix (no tilde). The root of the path in an ACL is AFTER the environment in a REST API call URL; that is, only put the /{resource}/{key} portion of the URL in the path. ACLs without a resource path are not permitted.

Environment

The environment directive can contain a single environment or a list. If environment isn't explicitly specified, it will default to all environments.

Method

Available methods are find, search, save, and destroy; you can specify one method or a list of them. If method isn't explicitly specified, it will default to all methods.

Auth

Each REST API call is either unauthenticated or authenticated with an SSL certificate; most communications between puppet agent and puppet master are authenticated. The value of auth can't be a list; it must be "yes" (or "on"), "no" (or "off"), or "any."

Allow

The node or nodes allowed to access this type of request. Can be a hostname, a certificate common name, a list of hostnames/certnames, or * (which matches all nodes). If the path for this ACL was a regular expression, allow directives may include backreferences to captured groups (e.g. $1).

An ACL may include multiple allow directives, which has the same effect as a single allow directive with a list.

Behavior in 0.25.x through 2.7.0: No fine-grained globbing of hostnames/certnames is available in allow directives; you must specify exact host/certnames, or a single asterisk that matches everything.

Behavior in 2.7.1 and later: Hostnames/certnames can also be specified by regular expression. Unlike with path directives, you don't need to use a tilde; just use the slash-quoting used in languages like Perl and Ruby (e.g. allow /^[\w-]+.magpie.lan$/). Regular expression allow directives can include backreferences to regex paths with the standard $1, $2 etc. variables.

Nodes cannot be allowed by IP address, unless the node's IP address is also its certname.

Any nodes which aren't specifically allowed to access the resource will be denied.

Deny

A deny directive is syntactically permitted, but has no effect.

Matching ACLs to Requests

Puppet composes a full list of ACLs by combining auth.conf with a list of default ACLs. When a request is received, ACLs are tested in their order of appearance, and **matching will stop at the first ACL that matches the request.**

An ACL matches a request only if its path, environment, method, and authentication **all** match that of the request. These four elements are equal peers in determining the match.

Matching Paths

If an ACL's path does not start with a tilde and a space, it matches the beginning of the resource path; an ACL with the line:

```
path /file
```

...will match both /file_metadata and /file_content resources.

Regular expression paths don't have to match from the beginning of the resource path, but it's good practice to use positional anchors.

```
path ~ ^/catalog/([^/]+)$
method find
allow $1
```

Captured groups from a regex path are available in the allow directive. The ACL above will allow nodes to retrieve their own catalog but prevent them from accessing other catalogs.

Determining Whether a Request is Allowed

Once an ACL has been determined to match an incoming request, Puppet consults the allow directive(s). If the request was unauthenticated, reverse DNS is used to determine the requesting node's hostname; the request is allowed if that hostname is allowed. If the request was authenticated, the certificate common name is read from the SSL certificate, and the hostname is ignored; the request is allowed if that certname is allowed.

Consequences of ACL Matching Behavior

Since ACLs are matched in linear order, auth.conf has to be manually arranged with the most specific paths at the top and the least specific paths at the bottom, lest the whole search get short-circuited and the (usually restrictive) fallback rule be applied to every request. Furthermore, since the default ACLs required for normal Puppet functionality are appended to the ACLs read from auth.conf, **you must manually interleave copies of the default ACLs into your auth.conf if you are specifying *any* ACLs that are less specific than any of the default ACLs.**

Default ACLs

Puppet appends a list of default ACLs to the ACLs read from auth.conf. However, if any custom ACLs have a path identical to that of a default ACL, that default ACL will be omitted when composing the full list of ACLs. **Note that this is not the same criteria for determining whether the two ACLs match the same requests,** as only the paths are compared:

```
# A custom ACL
path /file
auth no
allow magpie.lan

# This default ACL will not be appended to the rules
path /file
allow *
```

These two ACLs match completely disjoint sets of requests (unauthenticated for the custom one, authenticated for the default one), but since the mechanism that appends default ACLs is not examining the authentication/methods/environments of the ACLs, it considers the one to override the other, even though they're

effectively unrelated. Puppet agent will break if you only declare the custom ACL, will work if you manually declare the default ACL, and will work if you only declare the custom one but change its path to `/fil`. When in doubt, manually re-declare all default ACLs in your auth.conf file.

The following is a list of the default ACLs used by Puppet:

```
# Allow authenticated nodes to retrieve their own catalogs:

path ~ ^/catalog/([^/]+)$
method find
allow $1

# Allow authenticated nodes to access any file services --- in practice, this results in fileserver.conf
    being consulted:

path /file
allow *

# Allow authenticated nodes to access the certificate revocation list:

path /certificate_revocation_list/ca
method find
allow *

# Allow authenticated nodes to send reports:

path /report
method save
allow *

# Allow unauthenticated access to certificates:

path /certificate/ca
auth no
method find
allow *

path /certificate/
auth no
method find
allow *

# Allow unauthenticated nodes to submit certificate signing requests:

path /certificate_request
auth no
method find, save
allow *

# Deny all other requests:

path /
auth any
```

An example auth.conf file containing these rules is provided in the Puppet source, in conf/auth.conf.

Danger Mode

If you want to test the REST API for application prototyping without worrying about specifying your final set of ACLs ahead of time, you can set a completely permissive auth.conf:

```
path /
auth any
allow *
```

authconfig / namespaceauth.conf

Older versions of Puppet communicated over an XMLRPC interface instead of the current RESTful interface, and access to these APIs was governed by a file known as authconfig (after the configuration option listing its

location) or namespaceauth.conf (after its default filename). This legacy file will not be fully documented, but an example namespaceauth.conf file can be found in the puppet source at conf/namespaceauth.conf.

puppet agent and the REST API

If started with the listen = true configuration option, puppet agent will accept incoming REST API requests. This is most frequently used to trigger puppet runs with the run endpoint. Several caveats apply:

- A known bug in the 2.6.x releases of Puppet prevents puppet agent from being started with the listen = true option unless namespaceauth.conf is present, even though the file is never consulted. The workaround is to create an empty file: # touch $(puppet agent −−configprint authconfig)
- Puppet agent uses the same default ACLs as puppet master, which allow access to several useless endpoints while denying access to any agent-specific ones. For best results, you should short-circuit the defaults by denying access to / at the end of your auth.conf file.

Chapter 57

Type Reference

This page is autogenerated; any changes will get overwritten *(last generated on Sat May 21 17:20:18 -0700 2011)*

Resource Types

- The *namevar* is the parameter used to uniquely identify a type instance. This is the parameter that gets assigned when a string is provided before the colon in a type declaration. In general, only developers will need to worry about which parameter is the namevar.

  In the following code:

  ```
  file { "/etc/passwd":
    owner => root,
    group => root,
    mode => 644
  }
  ```

  /etc/passwd is considered the title of the file object (used for things like dependency handling), and because path is the namevar for file, that string is assigned to the path parameter.

- *Parameters* determine the specific configuration of the instance. They either directly modify the system (internally, these are called properties) or they affect how the instance behaves (e.g., adding a search path for exec instances or determining recursion on file instances).

- *Providers* provide low-level functionality for a given resource type. This is usually in the form of calling out to external commands.

  When required binaries are specified for providers, fully qualifed paths indicate that the binary must exist at that specific path and unqualified binaries indicate that Puppet will search for the binary using the shell path.

- *Features* are abilities that some providers might not support. You can use the list of supported features to determine how a given provider can be used.

  Resource types define features they can use, and providers can be tested to see which features they provide.

---

augeas

Apply the changes (single or array of changes) to the filesystem via the augeas tool.

Requires:

- augeas to be installed (http://www.augeas.net)

- ruby-augeas bindings

Sample usage with a string:

```
augeas{"test1" :
  context => "/files/etc/sysconfig/firstboot",
  changes => "set RUN_FIRSTBOOT YES",
  onlyif  => "match other_value size > 0",
}
```

Sample usage with an array and custom lenses:

```
augeas{"jboss_conf":
  context => "/files",
  changes => [
    "set /etc/jbossas/jbossas.conf/JBOSS_IP $ipaddress",
    "set /etc/jbossas/jbossas.conf/JAVA_HOME /usr"
  ],
  load_path => "$/usr/share/jbossas/lenses",
}
```

Features

- *execute_changes*: Actually make the changes
- *need_to_run?*: If the command should run
- *parse_commands*: Parse the command string

Provider execute_changes need_to_run? parse_commands augeas *X X X*

Parameters

changes The changes which should be applied to the filesystem. This can be either a string which contains a command or an array of commands. Commands supported are:

```
set [PATH] [VALUE]              Sets the value VALUE at loction PATH
rm [PATH]                       Removes the node at location PATH
remove [PATH]                   Synonym for rm
clear [PATH]                    Keeps the node at PATH, but removes the value.
ins [LABEL] [WHERE] [PATH]      Inserts an empty node LABEL either [WHERE={before|after}] PATH.
insert [LABEL] [WHERE] [PATH]   Synonym for ins
```

If the parameter 'context' is set that value is prepended to PATH

context Optional context path. This value is prepended to the paths of all changes if the path is relative. If INCL is set, defaults to '/files' + INCL, otherwise the empty string

force Optional command to force the augeas type to execute even if it thinks changes will not be made. This does not overide the only setting. If onlyif is set, then the foce setting will not override that result

incl Load only a specific file, e.g. /etc/hosts. When this parameter is set, you must also set the lens parameter to indicate which lens to use.

lens Use a specific lens, e.g. Hosts.lns. When this parameter is set, you must also set the incl parameter to indicate which file to load. Only that file will be loaded, which greatly speeds up execution of the type

load_path Optional colon separated list of directories; these directories are searched for schema definitions

name The name of this task. Used for uniqueness

onlyif Optional augeas command and comparisons to control the execution of this type. Supported onlyif syntax:

```
get [AUGEAS_PATH] [COMPARATOR] [STRING]
match [MATCH_PATH] size [COMPARATOR] [INT]
match [MATCH_PATH] include [STRING]
match [MATCH_PATH] not_include [STRING]
match [MATCH_PATH] == [AN_ARRAY]
match [MATCH_PATH] != [AN_ARRAY]
```

where:

```
AUGEAS_PATH is a valid path scoped by the context
MATCH_PATH is a valid match synatx scoped by the context
COMPARATOR is in the set [> >= != == <= <]
STRING is a string
INT is a number
AN_ARRAY is in the form ['a string', 'another']
```

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **augeas**: Supported features: execute_changes, need_to_run?, parse_commands.

returns The expected return code from the augeas command. Should not be set

root A file system path; all files loaded by Augeas are loaded underneath ROOT

type_check Set to true if augeas should perform typechecking. Optional, defaults to false Valid values are true, false.

---

computer

Computer object management using DirectoryService on OS X.

Note that these are distinctly different kinds of objects to 'hosts', as they require a MAC address and can have all sorts of policy attached to them.

This provider only manages Computer objects in the local directory service domain, not in remote directories.

If you wish to manage /etc/hosts file on Mac OS X, then simply use the host type as per other platforms.

This type primarily exists to create localhost Computer objects that MCX policy can then be attached to.

Autorequires: If Puppet is managing the plist file representing a Computer object (located at /var/db/dslocal/nodes/Default/computers/{name}.plist), the Computer resource will autorequire it.

Parameters

en_address The MAC address of the primary network interface. Must match en0.

ensure Control the existences of this computer record. Set this attribute to present to ensure the computer record exists. Set it to absent to delete any computer records with this name Valid values are present, absent.

ip_address The IP Address of the Computer object.

name The authoritative 'short' name of the computer record.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **directoryservice**: Computer object management using DirectoryService on OS X. Note that these are distinctly different kinds of objects to 'hosts', as they require a MAC address and can have all sorts of policy attached to them.

  This provider only manages Computer objects in the local directory service domain, not in remote directories.

  If you wish to manage /etc/hosts on Mac OS X, then simply use the host type as per other platforms. Default for operatingsystem == darwin.

realname The 'long' name of the computer record.

---

cron

Installs and manages cron jobs. All fields except the command and the user are optional, although specifying no periodic fields would result in the command being executed every minute. While the name of the cron job is not part of the actual job, it is used by Puppet to store and retrieve it.

If you specify a cron job that matches an existing job in every way except name, then the jobs will be considered equivalent and the new name will be permanently associated with that job. Once this association is made and synced to disk, you can then manage the job normally (e.g., change the schedule of the job).

Example:

```
cron { logrotate:
  command => "/usr/sbin/logrotate",
  user => root,
  hour => 2,
  minute => 0
}
```

Note that all cron values can be specified as an array of values:

```
cron { logrotate:
  command => "/usr/sbin/logrotate",
  user => root,
  hour => [2, 4]
}
```

Or using ranges, or the step syntax */2 (although there's no guarantee that your cron daemon supports it):

```
cron { logrotate:
  command => "/usr/sbin/logrotate",
  user => root,
  hour => ['2-4'],
  minute => '*/10'
}
```

Parameters

command The command to execute in the cron job. The environment provided to the command varies by local system rules, and it is best to always provide a fully qualified command. The user's profile is not sourced when the command is run, so if the user's environment is desired it should be sourced manually.

All cron parameters support absent as a value; this will remove any existing values for that field.

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

environment Any environment settings associated with this cron job. They will be stored between the header and the job in the crontab. There can be no guarantees that other, earlier settings will not also affect a given cron job.

Also, Puppet cannot automatically determine whether an existing, unmanaged environment setting is associated with a given cron job. If you already have cron jobs with environment settings, then Puppet will keep those settings in the same place in the file, but will not associate them with a specific job.

Settings should be specified exactly as they should appear in the crontab, e.g., PATH=/bin:/usr/bin:/usr/sbin.

hour The hour at which to run the cron job. Optional; if specified, must be between 0 and 23, inclusive.

minute The minute at which to run the cron job. Optional; if specified, must be between 0 and 59, inclusive.

month The month of the year. Optional; if specified must be between 1 and 12 or the month name (e.g., December).

monthday The day of the month on which to run the command. Optional; if specified, must be between 1 and 31.

name The symbolic name of the cron job. This name is used for human reference only and is generated automatically for cron jobs found on the system. This generally won't matter, as Puppet will do its best to match existing cron jobs against specified jobs (and Puppet adds a comment to cron jobs it adds), but it is at least possible that converting from unmanaged jobs to managed jobs might require manual intervention.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **crontab**: Required binaries: crontab.

special A special value such as 'reboot' or 'annually'. Only available on supported systems such as Vixie Cron. Overrides more specific time of day/week settings.

target Where the cron job should be stored. For crontab-style entries this is the same as the user and defaults that way. Other providers default accordingly.

user The user to run the command as. This user must be allowed to run cron jobs, which is not currently checked by Puppet.

The user defaults to whomever Puppet is running as.

weekday The weekday on which to run the command. Optional; if specified, must be between 0 and 7, inclusive, with 0 (or 7) being Sunday, or must be the name of the day (e.g., Tuesday).

---

exec

Executes external commands. It is critical that all commands executed using this mechanism can be run multiple times without harm, i.e., they are *idempotent*. One useful way to create idempotent commands is to use the checks like creates to avoid running the command unless some condition is met.

Note that you can restrict an exec to only run when it receives events by using the refreshonly parameter; this is a useful way to have your configuration respond to events with arbitrary commands.

Note also that if an exec receives an event from another resource, it will get executed again (or execute the command specified in refresh, if there is one).

There is a strong tendency to use exec to do whatever work Puppet can't already do; while this is obviously acceptable (and unavoidable) in the short term, it is highly recommended to migrate work from exec to native Puppet types as quickly as possible. If you find that you are doing a lot of work with exec, please at least notify us at Puppet Labs what you are doing, and hopefully we can work with you to get a native resource type for the work you are doing.

Autorequires: If Puppet is managing an exec's cwd or the executable file used in an exec's command, the exec resource will autorequire those files. If Puppet is managing the user that an exec should run as, the exec resource will autorequire that user.

Parameters

command

- **namevar**

The actual command to execute. Must either be fully qualified or a search path for the command must be provided. If the command succeeds, any output produced will be logged at the instance's normal log level (usually notice), but if the command fails (meaning its return code does not match the specified code) then any output is logged at the err log level.

creates A file that this command creates. If this parameter is provided, then the command will only be run if the specified file does not exist:

```
exec { "tar xf /my/tar/file.tar":
  cwd => "/var/tmp",
  creates => "/var/tmp/myfile",
  path => ["/usr/bin", "/usr/sbin"]
}
```

cwd The directory from which to run the command. If this directory does not exist, the command will fail.

environment Any additional environment variables you want to set for a command. Note that if you use this to set PATH, it will override the path attribute. Multiple environment variables should be specified as an array.

group The group to run the command as. This seems to work quite haphazardly on different platforms – it is a platform issue not a Ruby or Puppet one, since the same variety exists when running commnands as different users in the shell.

logoutput Whether to log output. Defaults to logging output at the loglevel for the exec resource. Use *on_failure* to only log the output when the command reports an error. Values are **true**, *false*, *on_failure*, and any legal log level. Valid values are true, false, on_failure.

onlyif If this parameter is set, then this exec will only run if the command returns 0. For example:

```
exec { "logrotate":
  path => "/usr/bin:/usr/sbin:/bin",
  onlyif => "test `du /var/log/messages | cut -f1` -gt 100000"
}
```

This would run logrotate only if that test returned true.

Note that this command follows the same rules as the main command, which is to say that it must be fully qualified if the path is not set.

Also note that onlyif can take an array as its value, e.g.:

```
onlyif => ["test -f /tmp/file1", "test -f /tmp/file2"]
```

This will only run the exec if /all/ conditions in the array return true.

path The search path used for command execution. Commands must be fully qualified if no path is specified. Paths can be specified as an array or as a colon separated list.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **posix**: Execute external binaries directly, on POSIX systems. This does not pass through a shell, or perform any interpolation, but only directly calls the command with the arguments given. Default for feature == posix.
- **shell**: Execute external binaries directly, on POSIX systems. passing through a shell so that shell built ins are available.

refresh How to refresh this command. By default, the exec is just called again when it receives an event from another resource, but this parameter allows you to define a different command for refreshing.

refreshonly The command should only be run as a refresh mechanism for when a dependent object is changed. It only makes sense to use this option when this command depends on some other object; it is useful for triggering an action:

```
# Pull down the main aliases file
file { "/etc/aliases":
  source => "puppet://server/module/aliases"
}

# Rebuild the database, but only when the file changes
exec { newaliases:
  path => ["/usr/bin", "/usr/sbin"],
  subscribe => File["/etc/aliases"],
  refreshonly => true
}
```

Note that only subscribe and notify can trigger actions, not require, so it only makes sense to use refreshonly with subscribe or notify. Valid values are true, false.

returns The expected return code(s). An error will be returned if the executed command returns something else. Defaults to 0. Can be specified as an array of acceptable return codes or a single value.

timeout The maximum time the command should take. If the command takes longer than the timeout, the command is considered to have failed and will be stopped. Use 0 to disable the timeout. The time is specified in seconds.

tries The number of times execution of the command should be tried. Defaults to '1'. This many attempts will be made to execute the command until an acceptable return code is returned. Note that the timeout paramater applies to each try rather than to the complete set of tries.

try_sleep The time to sleep in seconds between 'tries'.

unless If this parameter is set, then this exec will run unless the command returns 0. For example:

```
exec { "/bin/echo root >> /usr/lib/cron/cron.allow":
  path => "/usr/bin:/usr/sbin:/bin",
  unless => "grep root /usr/lib/cron/cron.allow 2>/dev/null"
}
```

This would add root to the cron.allow file (on Solaris) unless grep determines it's already there.

Note that this command follows the same rules as the main command, which is to say that it must be fully qualified if the path is not set.

user The user to run the command as. Note that if you use this then any error output is not currently captured. This is because of a bug within Ruby. If you are using Puppet to create this user, the exec will automatically require the user, as long as it is specified by name.

---

file

Manages local files, including setting ownership and permissions, creation of both files and directories, and retrieving entire files from remote servers. As Puppet matures, it expected that the file resource will be used less and less to manage content, and instead native resources will be used to do so.

If you find that you are often copying files in from a central location, rather than using native resources, please contact Puppet Labs and we can hopefully work with you to develop a native resource to support what you are doing.

Autorequires: If Puppet is managing the user or group that owns a file, the file resource will autorequire them. If Puppet is managing any parent directories of a file, the file resource will autorequire them.

Parameters

backup Whether files should be backed up before being replaced. The preferred method of backing files up is via a filebucket, which stores files by their MD5 sums and allows easy retrieval without littering directories with backups. You can specify a local filebucket or a network-accessible server-based filebucket by setting backup => bucket-name. Alternatively, if you specify any value that begins with a . (e.g., .puppet-bak), then Puppet will use copy the file in the same directory with that value as the extension of the backup. Setting backup => false disables all backups of the file in question.

Puppet automatically creates a local filebucket named puppet and defaults to backing up there. To use a server-based filebucket, you must specify one in your configuration

```
filebucket { main:
  server => puppet
}
```

The puppet master daemon creates a filebucket by default, so you can usually back up to your main server with this configuration. Once you've described the bucket in your configuration, you can use it in any file

```
file { "/my/file":
  source => "/path/in/nfs/or/something",
  backup => main
}
```

This will back the file up to the central server.

At this point, the benefits of using a filebucket are that you do not have backup files lying around on each of your machines, a given version of a file is only backed up once, and you can restore any given file manually, no matter how old. Eventually, transactional support will be able to automatically restore filebucketed files.

checksum The checksum type to use when checksumming a file.

The default checksum parameter, if checksums are enabled, is md5. Valid values are md5, md5lite, mtime, ctime, none.

content Specify the contents of a file as a string. Newlines, tabs, and spaces can be specified using the escaped syntax (e.g., \n for a newline). The primary purpose of this parameter is to provide a kind of limited templating:

```
define resolve(nameserver1, nameserver2, domain, search) {
    $str = "search $search
        domain $domain
        nameserver $nameserver1
        nameserver $nameserver2
        "
```

```
    file { "/etc/resolv.conf":
      content => $str
    }
}
```

This attribute is especially useful when used with templating.

ctime A read-only state to check the file ctime.

ensure Whether to create files that don't currently exist. Possible values are *absent*, *present*, *file*, and *directory*. Specifying present will match any form of file existence, and if the file is missing will create an empty file. Specifying absent will delete the file (and directory if recurse => true).

Anything other than those values will create a symlink. In the interest of readability and clarity, you should use ensure => link and explicitly specify a target; however, if a target attribute isn't provided, the value of the ensure attribute will be used as the symlink target:

```
# (Useful on Solaris)
# Less maintainable:
file { "/etc/inetd.conf":
  ensure => "/etc/inet/inetd.conf",
}

# More maintainable:
file { "/etc/inetd.conf":
  ensure => link,
  target => "/etc/inet/inetd.conf",
}
```

These two declarations are equivalent. Valid values are absent (also called false), file, present, directory, link. Values can match /./.

force Force the file operation. Currently only used when replacing directories with links. Valid values are true, false.

group Which group should own the file. Argument can be either group name or group ID.

ignore A parameter which omits action on files matching specified patterns during recursion. Uses Ruby's builtin globbing engine, so shell metacharacters are fully supported, e.g. [a-z]*. Matches that would descend into the directory structure are ignored, e.g., */*.

links How to handle links during file actions. During file copying, follow will copy the target file instead of the link, manage will copy the link itself, and ignore will just pass it by. When not copying, manage and ignore behave equivalently (because you cannot really ignore links entirely during local recursion), and follow will manage the file to which the link points. Valid values are follow, manage.

mode Mode the file should be. Currently relatively limited: you must specify the exact mode the file should be.

Note that when you set the mode of a directory, Puppet always sets the search/traverse (1) bit anywhere the read (4) bit is set. This is almost always what you want: read allows you to list the entries in a directory, and search/traverse allows you to access (read/write/execute) those entries.) Because of this feature, you can recursively make a directory and all of the files in it world-readable by setting e.g.:

```
file { '/some/dir':
  mode => 644,
  recurse => true,
}
```

In this case all of the files underneath /some/dir will have mode 644, and all of the directories will have mode 755.

mtime A read-only state to check the file mtime.

owner To whom the file should belong. Argument can be user name or user ID.

path

- **namevar**

The path to the file to manage. Must be fully qualified.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **microsoft_windows**: Uses Microsoft Windows functionality to manage file's users and rights.
- **posix**: Uses POSIX functionality to manage file's users and rights.

purge Whether unmanaged files should be purged. If you have a filebucket configured the purged files will be uploaded, but if you do not, this will destroy data. Only use this option for generated files unless you really know what you are doing. This option only makes sense when recursively managing directories.

Note that when using purge with source, Puppet will purge any files that are not on the remote system. Valid values are true, false.

recurse Whether and how deeply to do recursive management. Options are:

- inf,true — Regular style recursion on both remote and local directory structure.
- remote — Descends recursively into the remote directory but not the local directory. Allows copying of a few files into a directory containing many unmanaged files without scanning all the local files.
- false — Default of no recursion.
- [0−9]+ — Same as true, but limit recursion. Warning: this syntax has been deprecated in favor of the recurselimit attribute. Valid values are true, false, inf, remote. Values can match /^[0−9]+$/.

recurselimit How deeply to do recursive management. Values can match /^[0−9]+$/.

replace Whether or not to replace a file that is sourced but exists. This is useful for using file sources purely for initialization. Valid values are true (also called yes), false (also called no).

selinux_ignore_defaults If this is set then Puppet will not ask SELinux (via matchpathcon) to supply defaults for the SELinux attributes (seluser, selrole, seltype, and selrange). In general, you should leave this set at its default and only set it to true when you need Puppet to not try to fix SELinux labels automatically. Valid values are true, false.

selrange What the SELinux range component of the context of the file should be. Any valid SELinux range component is accepted. For example s0 or SystemHigh. If not specified it defaults to the value returned by matchpathcon for the file, if any exists. Only valid on systems with SELinux support enabled and that have support for MCS (Multi-Category Security).

selrole What the SELinux role component of the context of the file should be. Any valid SELinux role component is accepted. For example role_r. If not specified it defaults to the value returned by matchpathcon for the file, if any exists. Only valid on systems with SELinux support enabled.

seltype What the SELinux type component of the context of the file should be. Any valid SELinux type component is accepted. For example tmp_t. If not specified it defaults to the value returned by matchpathcon for the file, if any exists. Only valid on systems with SELinux support enabled.

seluser What the SELinux user component of the context of the file should be. Any valid SELinux user component is accepted. For example user_u. If not specified it defaults to the value returned by matchpathcon for the file, if any exists. Only valid on systems with SELinux support enabled.

source Copy a file over the current file. Uses checksum to determine when a file should be copied. Valid values are either fully qualified paths to files, or URIs. Currently supported URI types are *puppet* and *file*.

This is one of the primary mechanisms for getting content into applications that Puppet does not directly support and is very useful for those configuration files that don't change much across sytems. For instance:

```
class sendmail {
  file { "/etc/mail/sendmail.cf":
    source => "puppet://server/modules/module_name/sendmail.cf"
  }
}
```

You can also leave out the server name, in which case puppet agent will fill in the name of its configuration server and puppet apply will use the local filesystem. This makes it easy to use the same configuration in both local and centralized forms.

Currently, only the puppet scheme is supported for source URL's. Puppet will connect to the file server running on server to retrieve the contents of the file. If the server part is empty, the behavior of the command-line interpreter (puppet apply) and the client demon (puppet agent) differs slightly: apply will look such a file up on the module path on the local host, whereas agent will connect to the puppet server that it received the manifest from.

See the fileserver configuration documentation for information on how to configure and use file services within Puppet.

If you specify multiple file sources for a file, then the first source that exists will be used. This allows you to specify what amount to search paths for files:

```
file { "/path/to/my/file":
  source => [
    "/modules/nfs/files/file.$host",
    "/modules/nfs/files/file.$operatingsystem",
    "/modules/nfs/files/file"
  ]
}
```

This will use the first found file as the source.

You cannot currently copy links using this mechanism; set links to follow if any remote sources are links.

sourceselect Whether to copy all valid sources, or just the first one. This parameter is only used in recursive copies; by default, the first valid source is the only one used as a recursive source, but if this parameter is set to all, then all valid sources will have all of their contents copied to the local host, and for sources that have the same file, the source earlier in the list will be used. Valid values are first, all.

target The target for creating a link. Currently, symlinks are the only type supported.

You can make relative links:

```
# (Useful on Solaris)
file { "/etc/inetd.conf":
  ensure => link,
  target => "inet/inetd.conf",
}
```

You can also make recursive symlinks, which will create a directory structure that maps to the target directory, with directories corresponding to each directory and links corresponding to each file. Valid values are notlink. Values can match /./.

type A read-only state to check the file type.

---

filebucket

A repository for backing up files. If no filebucket is defined, then files will be backed up in their current directory, but the filebucket can be either a host- or site-global repository for backing up. It stores files and returns the MD5 sum, which can later be used to retrieve the file if restoration becomes necessary. A filebucket does not do any work itself; instead, it can be specified as the value of *backup* in a **file** object.

Currently, filebuckets are only useful for manual retrieval of accidentally removed files (e.g., you look in the log for the md5 sum and retrieve the file with that sum from the filebucket), but when transactions are fully supported filebuckets will be used to undo transactions.

You will normally want to define a single filebucket for your whole network and then use that as the default backup location:

```
# Define the bucket
filebucket { main: server => puppet }

# Specify it as the default target
File { backup => main }
```

Puppetmaster servers create a filebucket by default, so this will work in a default configuration.

Parameters

name The name of the filebucket.

path The path to the local filebucket. If this is unset, then the bucket is remote. The parameter *server* must can be specified to set the remote server.

port The port on which the remote server is listening. Defaults to the normal Puppet port, 8140.

server The server providing the remote filebucket. If this is not specified then *path* is checked. If it is set, then the bucket is local. Otherwise the puppetmaster server specified in the config or at the commandline is used.

---

group

Manage groups. On most platforms this can only create groups. Group membership must be managed on individual users.

On some platforms such as OS X, group membership is managed as an attribute of the group, not the user record. Providers must have the feature 'manages_members' to manage the 'members' property of a group record.

Features

- *manages_aix_lam*: The provider can manage AIX Loadable Authentication Module (LAM) system.
- *manages_members*: For directories where membership is an attribute of groups not users.
- *system_groups*: The provider allows you to create system groups with lower GIDs.

Provider manages_aix_lam manages_members system_groups aix *X* *X* directoryservice *X* groupadd *X* ldap pw

Parameters

allowdupe Whether to allow duplicate GIDs. This option does not work on FreeBSD (contract to the pw man page). Valid values are true, false.

attribute_membership Whether specified attribute value pairs should be treated as the only attributes of the user or whether they should merely be treated as the minimum list. Valid values are inclusive, minimum.

attributes Specify group AIX attributes in an array of keyvalue pairs Requires features manages_aix_lam.

auth_membership whether the provider is authoritative for group membership.

ensure Create or remove the group. Valid values are present, absent.

gid The group ID. Must be specified numerically. If not specified, a number will be picked, which can result in ID differences across systems and thus is not recommended. The GID is picked according to local system standards.

ia_load_module The name of the I&A module to use to manage this user Requires features manages_aix_lam.

members The members of the group. For directory services where group membership is stored in the group objects, not the users. Requires features manages_members.

name The group name. While naming limitations vary by system, it is advisable to keep the name to the degenerate limitations, which is a maximum of 8 characters beginning with a letter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **aix**: Group management for AIX! Users are managed with mkgroup, rmgroup, lsgroup, chgroup Required binaries: /usr/sbin/lsgroup, /usr/sbin/rmgroup, /usr/bin/chgroup, /usr/bin/mkgroup. Default for operatingsystem == aix. Supported features: manages_aix_lam, manages_members.
- **directoryservice**: Group management using DirectoryService on OS X.

  Required binaries: /usr/bin/dscl. Default for operatingsystem == darwin. Supported features: manages_members .
- **groupadd**: Group management via groupadd and its ilk.

  The default for most platforms

  Required binaries: groupdel, groupmod, groupadd. Supported features: system_groups.
- **ldap**: Group management via ldap.

  This provider requires that you have valid values for all of the ldap-related settings, including ldapbase. You will also almost definitely need settings for ldapuser and ldappassword, so that your clients can write to ldap.

  Note that this provider will automatically generate a GID for you if you do not specify one, but it is a potentially expensive operation, as it iterates across all existing groups to pick the appropriate next one.
- **pw**: Group management via pw.

  Only works on FreeBSD.

  Required binaries: /usr/sbin/pw. Default for operatingsystem == freebsd.

system Whether the group is a system group with lower GID. Valid values are true, false.

---

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

comment A comment that will be attached to the line with a # character

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.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **parsed**:

target The file in which to store service information. Only used by those providers that write to disk. On most systems this defaults to /etc/hosts.

---

interface

This represents a router or switch interface. It is possible to manage interface mode (access or trunking, native vlan and encapsulation), switchport characteristics (speed, duplex).

Parameters

allowed_trunk_vlans Allowed list of Vlans that this trunk can forward. Valid values are all. Values can match /./.

description Interface description.

device_url Url to connect to a router or switch.

duplex Interface duplex. Valid values are auto, full, half.

encapsulation Interface switchport encapsulation. Valid values are none, dot1q, isl.

ensure The basic property that the resource should be in. Valid values are present (also called no_shutdown), absent (also called shutdown).

etherchannel Channel group this interface is part of. Values can match /^\d+/.

ipaddress IP Address of this interface (it might not be possible to set an interface IP address it depends on the interface type and device type). Valid format of ip addresses are: * IPV4, like 127.0.0.1 * IPV4/prefixlength like 127.0.1.1/24 * IPV6/prefixlength like FE80::21A:2FFF:FE30:ECF0/128 * an optional suffix for IPV6 addresses from this list: eui–64, link-local It is also possible to use an array of values.

mode Interface switchport mode. Valid values are access, trunk.

name Interface name

native_vlan Interface native vlan (for access mode only). Values can match /^\d+/.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **cisco**: Cisco switch/router provider for interface.

speed Interface speed. Valid values are auto. Values can match /^\d+/.

---

k5login

Manage the .k5login file for a user. Specify the full path to the .k5login file as the name and an array of principals as the property principals.

Parameters

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

mode Manage the k5login file's mode

path

- **namevar**

The path to the file to manage. Must be fully qualified.

principals The principals present in the .k5login file.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **k5login**: The k5login provider is the only provider for the k5login type.

---

macauthorization

Manage the Mac OS X authorization database. See the Apple developer site for more information.

Autorequires: If Puppet is managing the /etc/authorization file, each macauthorization resource will autorequire it.

Parameters

allow_root Corresponds to 'allow-root' in the authorization store, renamed due to hyphens being problematic. Specifies whether a right should be allowed automatically if the requesting process is running with uid == 0. AuthorizationServices defaults this attribute to false if not specified Valid values are true, false.

auth_class Corresponds to 'class' in the authorization store, renamed due to 'class' being a reserved word. Valid values are user, evaluate−mechanisms, allow, deny, rule.

auth_type type - can be a 'right' or a 'rule'. 'comment' has not yet been implemented. Valid values are right, rule.

authenticate_user Corresponds to 'authenticate-user' in the authorization store, renamed due to hyphens being problematic. Valid values are true, false.

comment The 'comment' attribute for authorization resources.

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

group The user must authenticate as a member of this group. This attribute can be set to any one group.

k_of_n k-of-n describes how large a subset of rule mechanisms must succeed for successful authentication. If there are 'n' mechanisms, then 'k' (the integer value of this parameter) mechanisms must succeed. The most common setting for this parameter is '1'. If k-of-n is not set, then 'n-of-n' mechanisms must succeed.

mechanisms an array of suitable mechanisms.

name The name of the right or rule to be managed. Corresponds to 'key' in Authorization Services. The key is the name of a rule. A key uses the same naming conventions as a right. The Security Server uses a rule's key to match the rule with a right. Wildcard keys end with a '.'. The generic rule has an empty key value. Any rights that do not match a specific rule use the generic rule.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **macauthorization**: Manage Mac OS X authorization database rules and rights.
  Required binaries: /usr/bin/security, /usr/bin/sw_vers. Default for operatingsystem == darwin.

rule The rule(s) that this right refers to.

session_owner Corresponds to 'session-owner' in the authorization store, renamed due to hyphens being problematic. Whether the session owner automatically matches this rule or right. Valid values are true, false.

shared If this is set to true, then the Security Server marks the credentials used to gain this right as shared. The Security Server may use any shared credentials to authorize this right. For maximum security, set sharing to false so credentials stored by the Security Server for one application may not be used by another application. Valid values are true, false.

timeout The credential used by this rule expires in the specified number of seconds. For maximum security where the user must authenticate every time, set the timeout to 0. For minimum security, remove the timeout attribute so the user authenticates only once per session.

tries The number of tries allowed.

---

mailalias

Creates an email alias in the local alias database.

Parameters

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

name The alias name.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **aliases**:

recipient Where email should be sent. Multiple values should be specified as an array.

target The file in which to store the aliases. Only used by those providers that write to disk.

---

maillist

Manage email lists. This resource type currently can only create and remove lists, it cannot reconfigure them.

Parameters

admin The email address of the administrator.

description The description of the mailing list.

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

mailserver The name of the host handling email for the list.

name The name of the email list.

password The admin password.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **mailman**: Required binaries: /usr/lib/mailman/mail/mailman, /usr/lib/mailman/bin/list_lists, /usr/lib/mailman /bin/rmlist, /usr/lib/mailman/bin/newlist.

webserver The name of the host providing web archives and the administrative interface.

---

mcx

MCX object management using DirectoryService on OS X.

The default provider of this type merely manages the XML plist as reported by the dscl -mcxexport command. This is similar to the content property of the file type in Puppet.

The recommended method of using this type is to use Work Group Manager to manage users and groups on the local computer, record the resulting puppet manifest using the command puppet resource mcx, then deploy it to other machines.

Autorequires: If Puppet is managing the user, group, or computer that these MCX settings refer to, the MCX resource will autorequire that user, group, or computer.

Features

- *manages_content*: The provider can manage MCXSettings as a string.

Provider manages_content mcxcontent *X*

Parameters

content The XML Plist. The value of MCXSettings in DirectoryService. This is the standard output from the system command:

```
dscl localhost -mcxexport /Local/Default/<ds_type>/ds_name
```

Note that ds_type is capitalized and plural in the dscl command. Requires features manages_content.

ds_name The name to attach the MCX Setting to. e.g. 'localhost' when ds_type => computer. This setting is not required, as it may be parsed so long as the resource name is parseable. e.g. /Groups/admin where 'group' is the dstype.

ds_type The DirectoryService type this MCX setting attaches to. Valid values are user, group, computer, computerlist.

ensure Create or remove the MCX setting. Valid values are present, absent.

name The name of the resource being managed. The default naming convention follows Directory Service paths:

```
/Computers/localhost
/Groups/admin
/Users/localadmin
```

The ds_type and ds_name type parameters are not necessary if the default naming convention is followed.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **mcxcontent**: MCX Settings management using DirectoryService on OS X.

  This provider manages the entire MCXSettings attribute available to some directory services nodes. This management is 'all or nothing' in that discrete application domain key value pairs are not managed by this provider.

  It is recommended to use WorkGroup Manager to configure Users, Groups, Computers, or ComputerLists, then use 'ralsh mcx' to generate a puppet manifest from the resulting configuration.

  Original Author: Jeff McCune (mccune.jeff@gmail.com)

  Required binaries: /usr/bin/dscl. Default for operatingsystem == darwin. Supported features: manages_content .

---

mount

Manages mounted filesystems, including putting mount information into the mount table. The actual behavior depends on the value of the 'ensure' parameter.

Note that if a mount receives an event from another resource, it will try to remount the filesystems if ensure is set to mounted.

Features

- *refreshable*: The provider can remount the filesystem.

Provider refreshable parsed *X*

Parameters

atboot Whether to mount the mount at boot. Not all platforms support this.

blockdevice The device to fsck. This is property is only valid on Solaris, and in most cases will default to the correct value.

device The device providing the mount. This can be whatever device is supporting by the mount, including network devices or devices specified by UUID rather than device path, depending on the operating system.

dump Whether to dump the mount. Not all platform support this. Valid values are 1 or 0. or 2 on FreeBSD, Default is 0. Values can match /(0|1)/, /(0|1)/.

ensure Control what to do with this mount. Set this attribute to umounted to make sure the filesystem is in the filesystem table but not mounted (if the filesystem is currently mounted, it will be unmounted). Set it to absent to unmount (if necessary) and remove the filesystem from the fstab. Set to mounted to add it to the fstab and mount it. Set to present to add to fstab but not change mount/unmount status Valid values are defined (also called present), unmounted, absent, mounted.

fstype The mount type. Valid values depend on the operating system. This is a required option.

name The mount path for the mount.

options Mount options for the mounts, as they would appear in the fstab.

pass The pass in which the mount is checked.

path The deprecated name for the mount point. Please use name now.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **parsed**: Required binaries: mount, umount. Supported features: refreshable.

remounts Whether the mount can be remounted mount −o remount. If this is false, then the filesystem will be unmounted and remounted manually, which is prone to failure. Valid values are true, false.

target The file in which to store the mount table. Only used by those providers that write to disk.

---

nagios_command

The Nagios type command. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_command.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

command_line Nagios configuration file parameter.

command_name

- **namevar**

The name parameter for Nagios type command

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

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

target target

use Nagios configuration file parameter.

---

nagios_contact

The Nagios type contact. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_contact.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

address1 Nagios configuration file parameter.

address2 Nagios configuration file parameter.

address3 Nagios configuration file parameter.

address4 Nagios configuration file parameter.

address5 Nagios configuration file parameter.

address6 Nagios configuration file parameter.

alias Nagios configuration file parameter.

can_submit_commands Nagios configuration file parameter.

contact_name

- **namevar**

The name parameter for Nagios type contact

contactgroups Nagios configuration file parameter.

email Nagios configuration file parameter.

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

host_notification_commands Nagios configuration file parameter.

host_notification_options Nagios configuration file parameter.

host_notification_period Nagios configuration file parameter.

host_notifications_enabled Nagios configuration file parameter.

pager Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

retain_nonstatus_information Nagios configuration file parameter.

retain_status_information Nagios configuration file parameter.

service_notification_commands Nagios configuration file parameter.

service_notification_options Nagios configuration file parameter.

service_notification_period Nagios configuration file parameter.

service_notifications_enabled Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

---

nagios_contactgroup

The Nagios type contactgroup. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_contactgroup.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

alias Nagios configuration file parameter.

contactgroup_members Nagios configuration file parameter.

contactgroup_name

- **namevar**

The name parameter for Nagios type contactgroup

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

members Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

---

nagios_host

The Nagios type host. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_host.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

action_url Nagios configuration file parameter.

active_checks_enabled Nagios configuration file parameter.

address Nagios configuration file parameter.

alias Nagios configuration file parameter.

check_command Nagios configuration file parameter.

check_freshness Nagios configuration file parameter.

check_interval Nagios configuration file parameter.

check_period Nagios configuration file parameter.

contact_groups Nagios configuration file parameter.

contacts Nagios configuration file parameter.

display_name Nagios configuration file parameter.

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

event_handler Nagios configuration file parameter.

event_handler_enabled Nagios configuration file parameter.

failure_prediction_enabled Nagios configuration file parameter.

first_notification_delay Nagios configuration file parameter.

flap_detection_enabled Nagios configuration file parameter.

flap_detection_options Nagios configuration file parameter.

freshness_threshold Nagios configuration file parameter.

high_flap_threshold Nagios configuration file parameter.

host_name

- **namevar**

The name parameter for Nagios type host

hostgroups Nagios configuration file parameter.

icon_image Nagios configuration file parameter.

icon_image_alt Nagios configuration file parameter.

initial_state Nagios configuration file parameter.

low_flap_threshold Nagios configuration file parameter.

max_check_attempts Nagios configuration file parameter.

notes Nagios configuration file parameter.

notes_url Nagios configuration file parameter.

notification_interval Nagios configuration file parameter.

notification_options Nagios configuration file parameter.

notification_period Nagios configuration file parameter.

notifications_enabled Nagios configuration file parameter.

obsess_over_host Nagios configuration file parameter.

parents Nagios configuration file parameter.

passive_checks_enabled Nagios configuration file parameter.

process_perf_data Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

retain_nonstatus_information Nagios configuration file parameter.

retain_status_information Nagios configuration file parameter.

retry_interval Nagios configuration file parameter.

stalking_options Nagios configuration file parameter.

statusmap_image Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

vrml_image Nagios configuration file parameter.

---

nagios_hostdependency

The Nagios type hostdependency. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_hostdependency.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

_naginator_name

- **namevar**

The name parameter for Nagios type hostdependency

dependency_period Nagios configuration file parameter.

dependent_host_name Nagios configuration file parameter.

dependent_hostgroup_name Nagios configuration file parameter.

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

execution_failure_criteria Nagios configuration file parameter.

host_name Nagios configuration file parameter.

hostgroup_name Nagios configuration file parameter.

inherits_parent Nagios configuration file parameter.

notification_failure_criteria Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

---

nagios_hostescalation

The Nagios type hostescalation. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_hostescalation.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

_naginator_name

- **namevar**

The name parameter for Nagios type hostescalation

contact_groups Nagios configuration file parameter.

contacts Nagios configuration file parameter.

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

escalation_options Nagios configuration file parameter.

escalation_period Nagios configuration file parameter.

first_notification Nagios configuration file parameter.

host_name Nagios configuration file parameter.

hostgroup_name Nagios configuration file parameter.

last_notification Nagios configuration file parameter.

notification_interval Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

---

nagios_hostextinfo

The Nagios type hostextinfo. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_hostextinfo.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

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

host_name

- **namevar**

The name parameter for Nagios type hostextinfo

icon_image Nagios configuration file parameter.

icon_image_alt Nagios configuration file parameter.

notes Nagios configuration file parameter.

notes_url Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

statusmap_image Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

vrml_image Nagios configuration file parameter.

---

nagios_hostgroup

The Nagios type hostgroup. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_hostgroup.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

action_url Nagios configuration file parameter.

alias Nagios configuration file parameter.

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

hostgroup_members Nagios configuration file parameter.

hostgroup_name

- **namevar**

The name parameter for Nagios type hostgroup

members Nagios configuration file parameter.

notes Nagios configuration file parameter.

notes_url Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

---

nagios_service

The Nagios type service. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_service.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

_naginator_name

- **namevar**

The name parameter for Nagios type service

action_url Nagios configuration file parameter.

active_checks_enabled Nagios configuration file parameter.

check_command Nagios configuration file parameter.

check_freshness Nagios configuration file parameter.

check_interval Nagios configuration file parameter.

check_period Nagios configuration file parameter.

contact_groups Nagios configuration file parameter.

contacts Nagios configuration file parameter.

display_name Nagios configuration file parameter.

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

event_handler Nagios configuration file parameter.

event_handler_enabled Nagios configuration file parameter.

failure_prediction_enabled Nagios configuration file parameter.

first_notification_delay Nagios configuration file parameter.

flap_detection_enabled Nagios configuration file parameter.

flap_detection_options Nagios configuration file parameter.

freshness_threshold Nagios configuration file parameter.

high_flap_threshold Nagios configuration file parameter.

host_name Nagios configuration file parameter.

hostgroup_name Nagios configuration file parameter.

icon_image Nagios configuration file parameter.

icon_image_alt Nagios configuration file parameter.

initial_state Nagios configuration file parameter.

is_volatile Nagios configuration file parameter.

low_flap_threshold Nagios configuration file parameter.

max_check_attempts Nagios configuration file parameter.

normal_check_interval Nagios configuration file parameter.

notes Nagios configuration file parameter.

notes_url Nagios configuration file parameter.

notification_interval Nagios configuration file parameter.

notification_options Nagios configuration file parameter.

notification_period Nagios configuration file parameter.

notifications_enabled Nagios configuration file parameter.

obsess_over_service Nagios configuration file parameter.

parallelize_check Nagios configuration file parameter.

passive_checks_enabled Nagios configuration file parameter.

process_perf_data Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

retain_nonstatus_information Nagios configuration file parameter.

retain_status_information Nagios configuration file parameter.

retry_check_interval Nagios configuration file parameter.

retry_interval Nagios configuration file parameter.

service_description Nagios configuration file parameter.

servicegroups Nagios configuration file parameter.

stalking_options Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

---

nagios_servicedependency

The Nagios type servicedependency. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_servicedependency.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

_naginator_name

- **namevar**

The name parameter for Nagios type servicedependency

dependency_period Nagios configuration file parameter.

dependent_host_name Nagios configuration file parameter.

dependent_hostgroup_name Nagios configuration file parameter.

dependent_service_description Nagios configuration file parameter.

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

execution_failure_criteria Nagios configuration file parameter.

host_name Nagios configuration file parameter.

hostgroup_name Nagios configuration file parameter.

inherits_parent Nagios configuration file parameter.

notification_failure_criteria Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

service_description Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

---

nagios_serviceescalation

The Nagios type serviceescalation. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_serviceescalation.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

_naginator_name

- **namevar**

The name parameter for Nagios type serviceescalation

contact_groups Nagios configuration file parameter.

contacts Nagios configuration file parameter.

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

escalation_options Nagios configuration file parameter.

escalation_period Nagios configuration file parameter.

first_notification Nagios configuration file parameter.

host_name Nagios configuration file parameter.

hostgroup_name Nagios configuration file parameter.

last_notification Nagios configuration file parameter.

notification_interval Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

service_description Nagios configuration file parameter.

servicegroup_name Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

---

nagios_serviceextinfo

The Nagios type serviceextinfo. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_serviceextinfo.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

_naginator_name

- **namevar**

The name parameter for Nagios type serviceextinfo

action_url Nagios configuration file parameter.

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

host_name Nagios configuration file parameter.

icon_image Nagios configuration file parameter.

icon_image_alt Nagios configuration file parameter.

notes Nagios configuration file parameter.

notes_url Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

service_description Nagios configuration file parameter.

target target

use Nagios configuration file parameter.

---

nagios_servicegroup

The Nagios type servicegroup. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_servicegroup.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

action_url Nagios configuration file parameter.

alias Nagios configuration file parameter.

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

members Nagios configuration file parameter.

notes Nagios configuration file parameter.

notes_url Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

servicegroup_members Nagios configuration file parameter.

servicegroup_name

- **namevar**

The name parameter for Nagios type servicegroup

target target

use Nagios configuration file parameter.

---

nagios_timeperiod

The Nagios type timeperiod. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to /etc/nagios/nagios_timeperiod.cfg, but you can send them to a different file by setting their target attribute.

You can purge Nagios resources using the resources type, but *only* in the default file locations. This is an architectural limitation.

Parameters

alias Nagios configuration file parameter.

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

exclude Nagios configuration file parameter.

friday Nagios configuration file parameter.

monday Nagios configuration file parameter.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **naginator**:

register Nagios configuration file parameter.

saturday Nagios configuration file parameter.

sunday Nagios configuration file parameter.

target target

thursday Nagios configuration file parameter.

timeperiod_name

- **namevar**

The name parameter for Nagios type timeperiod

tuesday Nagios configuration file parameter.

use Nagios configuration file parameter.

wednesday Nagios configuration file parameter.

---

notify

Sends an arbitrary message to the agent run-time log.

Parameters

message The message to be sent to the log.

name An arbitrary tag for your own reference; the name of the message.

withpath Whether to not to show the full object path. Valid values are `true`, `false`.

---

package

Manage packages. There is a basic dichotomy in package support right now: Some package types (e.g., yum and apt) can retrieve their own package files, while others (e.g., rpm and sun) cannot. For those package formats that cannot retrieve their own files, you can use the `source` parameter to point to the correct file.

Puppet will automatically guess the packaging format that you are using based on the platform you are on, but you can override it using the `provider` parameter; each provider defines what it requires in order to function, and you must meet those requirements to use a given provider.

Autorequires: If Puppet is managing the files specified as a package's `adminfile`, `responsefile`, or `source`, the package resource will autorequire those files.

Features

- *holdable*: The provider is capable of placing packages on hold such that they are not automatically upgraded as a result of other package dependencies unless explicit action is taken by a user or another package. Held is considered a superset of installed.
- *installable*: The provider can install packages.
- *purgeable*: The provider can purge packages. This generally means that all traces of the package are removed, including existing configuration files. This feature is thus destructive and should be used with the utmost care.
- *uninstallable*: The provider can uninstall packages.
- *upgradeable*: The provider can upgrade to the latest version of a package. This feature is used by specifying `latest` as the desired value for the package.

- *versionable*: The provider is capable of interrogating the package database for installed version(s), and can select which out of a set of available versions of a package to install if asked.

Provider holdable installable purgeable uninstallable upgradeable versionable aix *X X X X* appdmg *X* apple *X* apt *X X X X X X* aptitude *X X X X X X* aptrpm *X X X X X* blastwave *X X X* dpkg *X X X X X* fink *X X X X X X* freebsd *X X* gem *X X X X* hpux *X X* macports *X X X X* nim *X X X X* openbsd *X X X* pip *X X X X* pkg *X X X* pkgdmg *X* pkgutil *X X X* portage *X X X X* ports *X X X* portupgrade *X X X* rpm *X X X X* rug *X X X X* sun *X X X* sunfreeware *X X X* up2date *X X X* urpmi *X X X X* yum *X X X X X* zypper *X X X X*

Parameters

adminfile A file containing package defaults for installing packages. This is currently only used on Solaris. The value will be validated according to system rules, which in the case of Solaris means that it should either be a fully qualified path or it should be in /var/sadm/install/admin.

allowcdrom Tells apt to allow cdrom sources in the sources.list file. Normally apt will bail if you try this. Valid values are true, false.

category A read-only parameter set by the package.

configfiles Whether configfiles should be kept or replaced. Most packages types do not support this parameter. Valid values are keep, replace.

description A read-only parameter set by the package.

ensure What state the package should be in. *latest* only makes sense for those packaging formats that can retrieve new packages on their own and will throw an error on those that cannot. For those packaging systems that allow you to specify package versions, specify them here. Similarly, *purged* is only useful for packaging systems that support the notion of managing configuration files separately from 'normal' system files. Valid values are present (also called installed), absent, purged, held, latest. Values can match /./.

flavor Newer versions of OpenBSD support 'flavors', which are further specifications for which type of package you want.

instance A read-only parameter set by the package.

name The package name. This is the name that the packaging system uses internally, which is sometimes (especially on Solaris) a name that is basically useless to humans. If you want to abstract package installation, then you can use aliases to provide a common name to packages:

```
# In the 'openssl' class
$ssl = $operatingsystem ? {
  solaris => SMCossl,
  default => openssl
}

# It is not an error to set an alias to the same value as the
# object name.
package { $ssl:
  ensure => installed,
  alias => openssl
}

. etc. .

$ssh = $operatingsystem ? {
  solaris => SMCossh,
  default => openssh
}
```

```
# Use the alias to specify a dependency, rather than
# having another selector to figure it out again.
package { $ssh:
  ensure => installed,
  alias => openssh,
  require => Package[openssl]
}
```

platform A read-only parameter set by the package.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **aix**: Installation from AIX Software directory Required binaries: /usr/bin/lslpp, /usr/sbin/installp. Default for operatingsystem == aix. Supported features: installable, uninstallable, upgradeable, versionable.
- **appdmg**: Package management which copies application bundles to a target. Required binaries: /usr/bin /hdiutil, /usr/bin/curl, /usr/bin/ditto. Supported features: installable.
- **apple**: Package management based on OS X's builtin packaging system. This is essentially the simplest and least functional package system in existence – it only supports installation; no deletion or upgrades. The provider will automatically add the .pkg extension, so leave that off when specifying the package name. Required binaries: /usr/sbin/installer. Supported features: installable.
- **apt**: Package management via apt−get. Required binaries: /usr/bin/apt−cache, /usr/bin/debconf−set−selections , /usr/bin/apt−get. Default for operatingsystem == debianubuntu. Supported features: holdable, installable, purgeable, uninstallable, upgradeable, versionable.
- **aptitude**: Package management via aptitude. Required binaries: /usr/bin/apt−cache, /usr/bin/aptitude. Supported features: holdable, installable, purgeable, uninstallable, upgradeable, versionable.
- **aptrpm**: Package management via apt−get ported to rpm. Required binaries: apt−cache, rpm, apt−get. Supported features: installable, purgeable, uninstallable, upgradeable, versionable.
- **blastwave**: Package management using Blastwave.org's pkg−get command on Solaris. Required binaries: pkg−get. Supported features: installable, uninstallable, upgradeable.
- **dpkg**: Package management via dpkg. Because this only uses dpkg and not apt, you must specify the source of any packages you want to manage. Required binaries: /usr/bin/dpkg−deb, /usr/bin/dpkg−query, /usr/bin/dpkg. Supported features: holdable, installable, purgeable, uninstallable, upgradeable.
- **fink**: Package management via fink. Required binaries: /sw/bin/apt−cache, /sw/bin/dpkg−query, /sw/bin/fink, /sw/bin/apt−get. Supported features: holdable, installable, purgeable, uninstallable, upgradeable, versionable.
- **freebsd**: The specific form of package management on FreeBSD. This is an extremely quirky packaging system, in that it freely mixes between ports and packages. Apparently all of the tools are written in Ruby, so there are plans to rewrite this support to directly use those libraries. Required binaries: /usr/ sbin/pkg_add, /usr/sbin/pkg_info, /usr/sbin/pkg_delete. Supported features: installable, uninstallable.
- **gem**: Ruby Gem support. If a URL is passed via source, then that URL is used as the remote gem repository; if a source is present but is not a valid URL, it will be interpreted as the path to a local gem file. If source is not present at all, the gem will be installed from the default gem repositories. Required binaries: gem. Supported features: installable, uninstallable, upgradeable, versionable.
- **hpux**: HP-UX's packaging system. Required binaries: /usr/sbin/swinstall, /usr/sbin/swlist, /usr/sbin/swremove . Default for operatingsystem == hp−ux. Supported features: installable, uninstallable.
- **macports**: Package management using MacPorts on OS X.

  Supports MacPorts versions and revisions, but not variants. Variant preferences may be specified using the MacPorts variants.conf file http://guide.macports.org/chunked/internals.configuration-files.html#internals.config files.variants-conf

  When specifying a version in the Puppet DSL, only specify the version, not the revision Revisions are only used internally for ensuring the latest version/revision of a port. Required binaries: /opt/local/bin/port. Supported features: installable, uninstallable, upgradeable, versionable.

- **nim**: Installation from NIM LPP source Required binaries: /usr/sbin/nimclient. Supported features: installable, uninstallable, upgradeable, versionable.
- **openbsd**: OpenBSD's form of pkg_add support. Required binaries: pkg_add, pkg_info, pkg_delete. Default for operatingsystem == openbsd. Supported features: installable, uninstallable, versionable.
- **pip**: Python packages via pip. Supported features: installable, uninstallable, upgradeable, versionable.
- **pkg**: OpenSolaris image packaging system. See pkg(5) for more information Required binaries: /usr/bin/pkg. Supported features: installable, uninstallable, upgradeable.
- **pkgdmg**: Package management based on Apple's Installer.app and DiskUtility.app. This package works by checking the contents of a DMG image for Apple pkg or mpkg files. Any number of pkg or mpkg files may exist in the root directory of the DMG file system. Sub directories are not checked for packages. See the wiki docs <http://projects.puppetlabs.com/projects/puppet/wiki/Package_Management_With_Dmg_Patterns> for more detail. Required binaries: /usr/bin/hdiutil, /usr/bin/curl, /usr/sbin/installer. Default for operatingsystem == darwin. Supported features: installable.
- **pkgutil**: Package management using Peter Bonivart's pkgutil command on Solaris. Required binaries: pkgutil. Supported features: installable, uninstallable, upgradeable.
- **portage**: Provides packaging support for Gentoo's portage system. Required binaries: /usr/bin/eix-update, /usr/bin/emerge, /usr/bin/eix. Default for operatingsystem == gentoo. Supported features: installable, uninstallable, upgradeable, versionable.
- **ports**: Support for FreeBSD's ports. Again, this still mixes packages and ports. Required binaries: /usr/local/sbin/portversion, /usr/local/sbin/pkg_deinstall, /usr/sbin/pkg_info, /usr/local/sbin/portupgrade. Default for operatingsystem == freebsd. Supported features: installable, uninstallable, upgradeable.
- **portupgrade**: Support for FreeBSD's ports using the portupgrade ports management software. Use the port's full origin as the resource name. eg (ports-mgmt/portupgrade) for the portupgrade port. Required binaries: /usr/local/sbin/portversion, /usr/local/sbin/pkg_deinstall, /usr/local/sbin/portinstall, /usr/sbin/pkg_info, /usr/local/sbin/portupgrade. Supported features: installable, uninstallable, upgradeable.
- **rpm**: RPM packaging support; should work anywhere with a working rpm binary. Required binaries: rpm. Supported features: installable, uninstallable, upgradeable, versionable.
- **rug**: Support for suse rug package manager. Required binaries: /usr/bin/rug, rpm. Default for operatingsystem == susesles. Supported features: installable, uninstallable, upgradeable, versionable.
- **sun**: Sun's packaging system. Requires that you specify the source for the packages you're managing. Required binaries: /usr/sbin/pkgadd, /usr/sbin/pkgrm, /usr/bin/pkginfo. Default for operatingsystem == solaris. Supported features: installable, uninstallable, upgradeable.
- **sunfreeware**: Package management using sunfreeware.com's pkg-get command on Solaris. At this point, support is exactly the same as blastwave support and has not actually been tested. Required binaries: pkg-get. Supported features: installable, uninstallable, upgradeable.
- **up2date**: Support for Red Hat's proprietary up2date package update mechanism. Required binaries: /usr/sbin/up2date-nox. Default for operatingsystem == redhatoelovm and lsbdistrelease == 2.134. Supported features: installable, uninstallable, upgradeable.
- **urpmi**: Support via urpmi. Required binaries: rpm, urpmi, urpmq. Default for operatingsystem == mandrivamandrake . Supported features: installable, uninstallable, upgradeable, versionable.
- **yum**: Support via yum. Required binaries: yum, python, rpm. Default for operatingsystem == fedoracentosredhat . Supported features: installable, purgeable, uninstallable, upgradeable, versionable.
- **zypper**: Support for SuSE zypper package manager. Found in SLES10sp2+ and SLES11 Required binaries: /usr/bin/zypper, rpm. Supported features: installable, uninstallable, upgradeable, versionable.

responsefile A file containing any necessary answers to questions asked by the package. This is currently used on Solaris and Debian. The value will be validated according to system rules, but it should generally be a fully qualified path.

root A read-only parameter set by the package.

source Where to find the actual package. This must be a local file (or on a network file system) or a URL that your specific packaging type understands; Puppet will not retrieve files for you.

status A read-only parameter set by the package.

type Deprecated form of provider.

vendor A read-only parameter set by the package.

---

resources

This is a metatype that can manage other resource types. Any metaparams specified here will be passed on to any generated resources, so you can purge umanaged resources but set noop to true so the purging is only logged and does not actually happen.

Parameters

name The name of the type to be managed.

purge Purge unmanaged resources. This will delete any resource that is not specified in your configuration and is not required by any specified resources. Valid values are true, false.

unless_system_user This keeps system users from being purged. By default, it does not purge users whose UIDs are less than or equal to 500, but you can specify a different UID as the inclusive limit. Valid values are true, false. Values can match /^\d+$/.

---

router

Manages connected router.

Parameters

url

- **namevar**

An URL to access the router of the form (ssh telnet)://user:pass:enable@host/.

---

schedule

Defined schedules for Puppet. The important thing to understand about how schedules are currently implemented in Puppet is that they can only be used to stop a resource from being applied, they never guarantee that it is applied.

Every time Puppet applies its configuration, it will collect the list of resources whose schedule does not eliminate them from running right then, but there is currently no system in place to guarantee that a given resource runs at a given time. If you specify a very restrictive schedule and Puppet happens to run at a time within that schedule, then the resources will get applied; otherwise, that work may never get done.

Thus, it behooves you to use wider scheduling (e.g., over a couple of hours) combined with periods and repetitions. For instance, if you wanted to restrict certain resources to only running once, between the hours of two and 4 AM, then you would use this schedule:

```
schedule { maint:
  range => "2 - 4",
  period => daily,
  repeat => 1
}
```

With this schedule, the first time that Puppet runs between 2 and 4 AM, all resources with this schedule will get applied, but they won't get applied again between 2 and 4 because they will have already run once that day, and they won't get applied outside that schedule because they will be outside the scheduled range.

Puppet automatically creates a schedule for each valid period with the same name as that period (e.g., hourly and daily). Additionally, a schedule named *puppet* is created and used as the default, with the following attributes:

```
schedule { puppet:
  period => hourly,
  repeat => 2
}
```

This will cause resources to be applied every 30 minutes by default.

Parameters

name The name of the schedule. This name is used to retrieve the schedule when assigning it to an object:

```
schedule { daily:
  period => daily,
  range => "2 - 4",
}

exec { "/usr/bin/apt-get update":
  schedule => daily
}
```

period The period of repetition for a resource. Choose from among a fixed list of *hourly*, *daily*, *weekly*, and *monthly*. The default is for a resource to get applied every time that Puppet runs, whatever that period is.

Note that the period defines how often a given resource will get applied but not when; if you would like to restrict the hours that a given resource can be applied (e.g., only at night during a maintenance window) then use the range attribute.

If the provided periods are not sufficient, you can provide a value to the *repeat* attribute, which will cause Puppet to schedule the affected resources evenly in the period the specified number of times. Take this schedule:

```
schedule { veryoften:
  period => hourly,
  repeat => 6
}
```

This can cause Puppet to apply that resource up to every 10 minutes.

At the moment, Puppet cannot guarantee that level of repetition; that is, it can run up to every 10 minutes, but internal factors might prevent it from actually running that often (e.g., long-running Puppet runs will squash conflictingly scheduled runs).

See the periodmatch attribute for tuning whether to match times by their distance apart or by their specific value. Valid values are hourly, daily, weekly, monthly, never.

periodmatch Whether periods should be matched by number (e.g., the two times are in the same hour) or by distance (e.g., the two times are 60 minutes apart). Valid values are number, distance.

range The earliest and latest that a resource can be applied. This is always a range within a 24 hour period, and hours must be specified in numbers between 0 and 23, inclusive. Minutes and seconds can be provided, using the normal colon as a separator. For instance:

```
schedule { maintenance:
  range => "1:30 - 4:30"
}
```

This is mostly useful for restricting certain resources to being applied in maintenance windows or during off-peak hours.

repeat How often the application gets repeated in a given period. Defaults to 1. Must be an integer.

---

selboolean

Manages SELinux booleans on systems with SELinux support. The supported booleans are any of the ones found in /selinux/booleans/.

Parameters

name The name of the SELinux boolean to be managed.

persistent If set true, SELinux booleans will be written to disk and persist accross reboots. The default is false. Valid values are true, false.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **getsetsebool**: Manage SELinux booleans using the getsebool and setsebool binaries. Required binaries: /usr/sbin/getsebool, /usr/sbin/setsebool.

value Whether the the SELinux boolean should be enabled or disabled. Valid values are on, off.

---

selmodule

Manages loading and unloading of SELinux policy modules on the system. Requires SELinux support. See man semodule(8) for more information on SELinux policy modules.

Autorequires: If Puppet is managing the file containing this SELinux policy module (which is either explicitly specified in the selmodulepath attribute or will be found at {selmoduledir}/{name}.pp), the selmodule resource will autorequire that file.

Parameters

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

name The name of the SELinux policy to be managed. You should not include the customary trailing .pp extension.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **semodule**: Manage SELinux policy modules using the semodule binary. Required binaries: /usr/sbin/semodule.

selmoduledir The directory to look for the compiled pp module file in. Currently defaults to /usr/share/selinux /targeted. If selmodulepath is not specified the module will be looked for in this directory in a in a file called NAME.pp, where NAME is the value of the name parameter.

selmodulepath The full path to the compiled .pp policy module. You only need to use this if the module file is not in the directory pointed at by selmoduledir.

syncversion If set to true, the policy will be reloaded if the version found in the on-disk file differs from the loaded version. If set to false (the default) the the only check that will be made is if the policy is loaded at all or not. Valid values are true, false.

---

service

Manage running services. Service support unfortunately varies widely by platform — some platforms have very little if any concept of a running service, and some have a very codified and powerful concept. Puppet's service support will generally be able to do the right thing regardless (e.g., if there is no 'status' command, then Puppet will look in the process table for a command matching the service name), but the more information you can provide, the better behaviour you will get. In particular, any virtual services that don't have a predictable entry in the process table (for example, network on Red Hat/CentOS systems) will manifest odd behavior on restarts if you don't specify hasstatus or a status command.

Note that if a service receives an event from another resource, the service will get restarted. The actual command to restart the service depends on the platform. You can provide an explicit command for restarting with the restart attribute, or use the init script's restart command with the hasrestart attribute; if you do neither, the service's stop and start commands will be used.

Features

- *controllable*: The provider uses a control variable.
- *enableable*: The provider can enable and disable the service
- *refreshable*: The provider can restart the service.

Provider controllable enableable refreshable base *X* bsd *X X* daemontools *X X* debian *X X* freebsd *X X* gentoo *X X* init *X* launchd *X X* redhat *X X* runit *X X* smf *X X* src *X* upstart *X*

Parameters

binary The path to the daemon. This is only used for systems that do not support init scripts. This binary will be used to start the service if no start parameter is provided.

control The control variable used to manage services (originally for HP-UX). Defaults to the upcased service name plus START replacing dots with underscores, for those providers that support the controllable feature.

enable Whether a service should be enabled to start at boot. This property behaves quite differently depending on the platform; wherever possible, it relies on local tools to enable or disable a given service. Valid values are true, false. Requires features enableable.

ensure Whether a service should be running. Valid values are stopped (also called false), running (also called true).

hasrestart Specify that an init script has a restart option. Otherwise, the init script's stop and start methods are used. Valid values are true, false.

hasstatus Declare the the service's init script has a functional status command. Based on testing, it was found that a large number of init scripts on different platforms do not support any kind of status command; thus, you must specify manually whether the service you are running has such a command. Alternately, you can provide a specific command using the status attribute.

If you specify neither of these, then Puppet will look for the service name in the process table. Be aware that 'virtual' init scripts such as networking will respond poorly to refresh events (via notify and subscribe relationships) if you don't override this default behavior. Valid values are true, false.

manifest Specify a command to config a service, or a path to a manifest to do so.

name The name of the service to run. This name is used to find the service in whatever service subsystem it is in.

path The search path for finding init scripts. Multiple values should be separated by colons or provided as an array.

pattern The pattern to search for in the process table. This is used for stopping services on platforms that do not support init scripts, and is also used for determining service status on those service whose init scripts do not include a status command.

If this is left unspecified and is needed to check the status of a service, then the service name will be used instead.

The pattern can be a simple string or any legal Ruby pattern.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **base**: The simplest form of service support.

  You have to specify enough about your service for this to work; the minimum you can specify is a binary for starting the process, and this same binary will be searched for in the process table to stop the service. It is preferable to specify start, stop, and status commands, akin to how you would do so using init.

  Required binaries: kill. Supported features: refreshable.

- **bsd**: FreeBSD's (and probably NetBSD?) form of init-style service management.

  Uses rc.conf.d for service enabling and disabling.

  Supported features: 'enableable', 'refreshable'.

- **daemontools**: Daemontools service management.

  This provider manages daemons running supervised by D.J.Bernstein daemontools. It tries to detect the service directory, with by order of preference:

  - /service
  - /etc/service
  - /var/lib/svscan

  The daemon directory should be placed in a directory that can be by default in:

  - /var/lib/service
  - /etc

  or this can be overriden in the service resource parameters::

  ```
  service { "myservice": provider => "daemontools", path => "/path/to/daemons",
  }
  ```

  This provider supports out of the box:

- start/stop (mapped to enable/disable)
- enable/disable
- restart
- status

If a service has ensure => "running", it will link /path/to/daemon to /path/to/service, which will automatically enable the service.

If a service has ensure => "stopped", it will only down the service, not remove the /path/to/service link.

Required binaries: /usr/bin/svc, /usr/bin/svstat. Supported features: enableable, refreshable.

- **debian**: Debian's form of init-style management.

  The only difference is that this supports service enabling and disabling via update-rc.d and determines enabled status via invoke-rc.d.

  Required binaries: /usr/sbin/update-rc.d, /usr/sbin/invoke-rc.d. Default for operatingsystem == debianubuntu. Supported features: enableable, refreshable.

- **freebsd**: Provider for FreeBSD. Makes use of rcvar argument of init scripts and parses/edits rc files. Default for operatingsystem == freebsd. Supported features: enableable, refreshable.

- **gentoo**: Gentoo's form of init-style service management.

  Uses rc-update for service enabling and disabling.

  Required binaries: /sbin/rc-update. Default for operatingsystem == gentoo. Supported features: enableable, refreshable.

- **init**: Standard init service management.

  This provider assumes that the init script has no status command, because so few scripts do, so you need to either provide a status command or specify via hasstatus that one already exists in the init script.

  ```
  Supported features: 'refreshable'.
  ```

- **launchd**: launchd service management framework.

  This provider manages jobs with launchd, which is the default service framework for Mac OS X and is potentially available for use on other platforms.

  See:

  - http://developer.apple.com/macosx/launchd.html
  - http://launchd.macosforge.org/

  This provider reads plists out of the following directories:

  - /System/Library/LaunchDaemons
  - /System/Library/LaunchAgents
  - /Library/LaunchDaemons
  - /Library/LaunchAgents

  ...and builds up a list of services based upon each plist's "Label" entry.

  This provider supports:

  - ensure => running/stopped,
  - enable => true/false
  - status
  - restart

  Here is how the Puppet states correspond to launchd states:

  - stopped — job unloaded

  - started — job loaded
  - enabled — 'Disable' removed from job plist file
  - disabled — 'Disable' added to job plist file

  Note that this allows you to do something launchctl can't do, which is to be in a state of "stopped/enabled or "running/disabled".

  Required binaries: /bin/launchctl, /usr/bin/plutil, /usr/bin/sw_vers. Default for operatingsystem == darwin. Supported features: enableable, refreshable.

- **redhat**: Red Hat's (and probably many others) form of init-style service management:

  Uses chkconfig for service enabling and disabling.

  Required binaries: /sbin/service, /sbin/chkconfig. Default for operatingsystem == redhatfedorasusecentosslesoelovm . Supported features: enableable, refreshable.

- **runit**: Runit service management.

  This provider manages daemons running supervised by Runit. It tries to detect the service directory, with by order of preference:

  - /service
  - /var/service
  - /etc/service

  The daemon directory should be placed in a directory that can be by default in:

  - /etc/sv

  or this can be overriden in the service resource parameters::

```
service { "myservice": provider => "runit", path => "/path/to/daemons",
}
```

  This provider supports out of the box:

  - start/stop
  - enable/disable
  - restart
  - status

  Required binaries: /usr/bin/sv. Supported features: enableable, refreshable.

- **smf**: Support for Sun's new Service Management Framework.

  Starting a service is effectively equivalent to enabling it, so there is only support for starting and stopping services, which also enables and disables them, respectively.

  By specifying manifest => "/path/to/service.xml", the SMF manifest will be imported if it does not exist.

  Required binaries: /usr/sbin/svcadm, /usr/bin/svcs, /usr/sbin/svccfg. Default for operatingsystem == solaris. Supported features: enableable, refreshable.

- **src**: Support for AIX's System Resource controller.

  Services are started/stopped based on the stopsrc and startsrc commands, and some services can be refreshed with refresh command.

  - Enabling and disableing services is not supported, as it requires modifications to /etc/inittab.
  - Starting and stopping groups of subsystems is not yet supported Required binaries: /usr/bin/stopsrc, /usr/bin/startsrc, /usr/bin/lssrc, /usr/bin/refresh. Default for operatingsystem == aix. Supported features: refreshable.

- **upstart**: Ubuntu service manager upstart.

  This provider manages upstart jobs which have replaced initd.

  See: * http://upstart.ubuntu.com/ Required binaries: /sbin/restart, /sbin/start, /sbin/status, /sbin/initctl, /sbin/stop. Supported features: refreshable.

restart Specify a *restart* command manually. If left unspecified, the service will be stopped and then started.

start Specify a *start* command manually. Most service subsystems support a start command, so this will not need to be specified.

status Specify a *status* command manually. This command must return 0 if the service is running and a nonzero value otherwise. Ideally, these return codes should conform to the LSB's specification for init script status actions, but puppet only considers the difference between 0 and nonzero to be relevant.

If left unspecified, the status method will be determined automatically, usually by looking for the service in the process table.

stop Specify a *stop* command manually.

---

ssh_authorized_key

Manages SSH authorized keys. Currently only type 2 keys are supported.

Autorequires: If Puppet is managing the user account in which this SSH key should be installed, the ssh_authorized_key resource will autorequire that user.

Parameters

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

key The key itself; generally a long string of hex digits.

name The SSH key comment. This attribute is currently used as a system-wide primary key and therefore has to be unique.

options Key options, see sshd(8) for possible values. Multiple values should be specified as an array.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **parsed**: Parse and generate authorized_keys files for SSH.

target The absolute filename in which to store the SSH key. This property is optional and should only be used in cases where keys are stored in a non-standard location (i.e. not in~user/.ssh/authorized_keys').

type The encryption type used: ssh-dss or ssh-rsa. Valid values are ssh−dss (also called dsa), ssh−rsa (also called rsa).

user The user account in which the SSH key should be installed. The resource will automatically depend on this user.

---

sshkey

Installs and manages ssh host keys. At this point, this type only knows how to install keys into /etc/ssh/ssh_known_hosts. See the ssh_authorized_key type to manage authorized keys.

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.

key The key itself; generally a long string of hex digits.

name The host name that the key is associated with.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **parsed**: Parse and generate host-wide known hosts files for SSH.

target The file in which to store the ssh key. Only used by the parsed provider.

type The encryption type used. Probably ssh-dss or ssh-rsa. Valid values are ssh−dss (also called dsa), ssh−rsa (also called rsa).

---

stage

A resource type for specifying run stages. The actual stage should be specified on resources:

```
class { foo: stage => pre }
```

And you must manually control stage order:

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

You automatically get a 'main' stage created, and by default all resources get inserted into that stage.

You can only set stages on class resources, not normal builtin resources.

Parameters

name The name of the stage. This will be used as the 'stage' for each resource.

---

tidy

Remove unwanted files based on specific criteria. Multiple criteria are OR'd together, so a file that is too large but is not old enough will still get tidied.

If you don't specify either age or size, then all files will be removed.

This resource type works by generating a file resource for every file that should be deleted and then letting that resource perform the actual deletion.

Parameters

age Tidy files whose age is equal to or greater than the specified time. You can choose seconds, minutes, hours, days, or weeks by specifying the first letter of any of those words (e.g., '1w').

Specifying 0 will remove all files.

backup Whether tidied files should be backed up. Any values are passed directly to the file resources used for actual file deletion, so use its backup documentation to determine valid values.

matches One or more (shell type) file glob patterns, which restrict the list of files to be tidied to those whose basenames match at least one of the patterns specified. Multiple patterns can be specified using an array.

Example:

```
tidy { "/tmp":
  age => "1w",
  recurse => 1,
  matches => [ "[0-9]pub*.tmp", "*.temp", "tmpfile?" ]
}
```

This removes files from /tmp if they are one week old or older, are not in a subdirectory and match one of the shell globs given.

Note that the patterns are matched against the basename of each file – that is, your glob patterns should not have any '/' characters in them, since you are only specifying against the last bit of the file.

Finally, note that you must now specify a non-zero/non-false value for recurse if matches is used, as matches only apply to files found by recursion (there's no reason to use static patterns match against a statically determined path). Requiering explicit recursion clears up a common source of confusion.

path

- **namevar**

The path to the file or directory to manage. Must be fully qualified.

recurse If target is a directory, recursively descend into the directory looking for files to tidy. Valid values are true, false, inf. Values can match /^[0-9]+$/.

rmdirs Tidy directories in addition to files; that is, remove directories whose age is older than the specified criteria. This will only remove empty directories, so all contained files must also be tidied before a directory gets removed. Valid values are true, false.

size Tidy files whose size is equal to or greater than the specified size. Unqualified values are in kilobytes, but *b*, *k*, *m*, *g*, and *t* can be appended to specify *bytes*, *kilobytes*, *megabytes*, *gigabytes*, and *terabytes*, respectively. Only the first character is significant, so the full word can also be used.

type Set the mechanism for determining age. Valid values are atime, mtime, ctime.

---

user

Manage users. This type is mostly built to manage system users, so it is lacking some features useful for managing normal users.

This resource type uses the prescribed native tools for creating groups and generally uses POSIX APIs for retrieving information about them. It does not directly modify /etc/passwd or anything.

Autorequires: If Puppet is managing the user's primary group (as provided in the gid attribute), the user resource will autorequire that group. If Puppet is managing any role accounts corresponding to the user's roles, the user resource will autorequire those role accounts.

Features

- *allows_duplicates*: The provider supports duplicate users with the same UID.
- *manages_aix_lam*: The provider can manage AIX Loadable Authentication Module (LAM) system.
- *manages_expiry*: The provider can manage the expiry date for a user.
- *manages_homedir*: The provider can create and remove home directories.
- *manages_password_age*: The provider can set age requirements and restrictions for passwords.
- *manages_passwords*: The provider can modify user passwords, by accepting a password hash.
- *manages_solaris_rbac*: The provider can manage roles and normal users
- *system_users*: The provider allows you to create system users with lower UIDs.

Provider allows_duplicates manages_aix_lam manages_expiry manages_homedir manages_password_age manages_passwords manages_solaris_rbac system_users aix *X X X X X* directoryservice *X* hpuxuseradd *X X* ldap *X* pw *X X* user_role_add *X X X X X* useradd *X X X X X X*

Parameters

allowdupe Whether to allow duplicate UIDs. Valid values are true, false.

attribute_membership Whether specified attribute value pairs should be treated as the only attributes of the user or whether they should merely be treated as the minimum list. Valid values are inclusive, minimum.

attributes Specify user AIX attributes in an array of keyvalue pairs Requires features manages_aix_lam.

auth_membership Whether specified auths should be treated as the only auths of which the user is a member or whether they should merely be treated as the minimum membership list. Valid values are inclusive, minimum.

auths The auths the user has. Multiple auths should be specified as an array. Requires features manages_solaris_rbac.

comment A description of the user. Generally is a user's full name.

ensure The basic state that the object should be in. Valid values are present, absent, role.

expiry The expiry date for this user. Must be provided in a zero padded YYYY-MM-DD format - e.g 2010–02–19. Requires features manages_expiry.

gid The user's primary group. Can be specified numerically or by name.

groups The groups of which the user is a member. The primary group should not be listed. Multiple groups should be specified as an array.

home The home directory of the user. The directory must be created separately and is not currently checked for existence.

ia_load_module The name of the I&A module to use to manage this user Requires features manages_aix_lam.

key_membership Whether specified key value pairs should be treated as the only attributes of the user or whether they should merely be treated as the minimum list. Valid values are inclusive, minimum.

keys Specify user attributes in an array of keyvalue pairs Requires features manages_solaris_rbac.

managehome Whether to manage the home directory when managing the user. Valid values are true, false.

membership Whether specified groups should be treated as the only groups of which the user is a member or whether they should merely be treated as the minimum membership list. Valid values are inclusive, minimum.

name User name. While limitations are determined for each operating system, it is generally a good idea to keep to the degenerate 8 characters, beginning with a letter.

password The user's password, in whatever encrypted format the local machine requires. Be sure to enclose any value that includes a dollar sign ($) in single quotes (‘). Requires features manages_passwords.

password_max_age The maximum amount of time in days a password may be used before it must be changed Requires features manages_password_age.

password_min_age The minimum amount of time in days a password must be used before it may be changed Requires features manages_password_age.

profile_membership Whether specified roles should be treated as the only roles of which the user is a member or whether they should merely be treated as the minimum membership list. Valid values are inclusive, minimum.

profiles The profiles the user has. Multiple profiles should be specified as an array. Requires features manages_solaris_rbac.

project The name of the project associated with a user Requires features manages_solaris_rbac.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **aix**: User management for AIX! Users are managed with mkuser, rmuser, chuser, lsuser Required binaries: /usr/sbin/lsgroup, /bin/chpasswd, /usr/sbin/lsuser, /usr/sbin/rmuser, /usr/bin/chuser, /usr/bin/mkuser. Default for operatingsystem == aix. Supported features: manages_aix_lam, manages_expiry, manages_homedir, manages_password_age, manages_passwords.
- **directoryservice**: User management using DirectoryService on OS X. Required binaries: /usr/bin/dscl. Default for operatingsystem == darwin. Supported features: manages_passwords.
- **hpuxuseradd**: User management for hp-ux! Undocumented switch to special usermod because HP-UX regular usermod is TOO STUPID to change stuff while the user is logged in. Required binaries: /usr/sam/lbin/userdel.sam, /usr/sam/lbin/usermod.sam, /usr/sbin/useradd. Default for operatingsystem == hp−ux. Supported features: allows_duplicates, manages_homedir.
- **ldap**: User management via ldap. This provider requires that you have valid values for all of the ldap-related settings, including ldapbase. You will also almost definitely need settings for ldapuser and ldappassword , so that your clients can write to ldap.

  Note that this provider will automatically generate a UID for you if you do not specify one, but it is a potentially expensive operation, as it iterates across all existing users to pick the appropriate next one. Supported features: manages_passwords.
- **pw**: User management via pw on FreeBSD. Required binaries: pw. Default for operatingsystem == freebsd. Supported features: allows_duplicates, manages_homedir.
- **user_role_add**: User management inherits useradd and adds logic to manage roles on Solaris using roleadd. Required binaries: roleadd, roledel, userdel, passwd, rolemod, usermod, useradd. Default for operatingsystem == solaris. Supported features: allows_duplicates, manages_homedir, manages_password_age, manages_passwords, manages_solaris_rbac.

- **useradd**: User management via useradd and its ilk. Note that you will need to install the Shadow Password Ruby library often known as ruby-libshadow to manage user passwords. Required binaries: userdel, chage, usermod, useradd. Supported features: allows_duplicates, manages_expiry, manages_homedir, manages_password_age, manages_passwords, system_users.

role_membership Whether specified roles should be treated as the only roles of which the user is a member or whether they should merely be treated as the minimum membership list. Valid values are inclusive, minimum.

roles The roles the user has. Multiple roles should be specified as an array. Requires features manages_solaris_rbac.

shell The user's login shell. The shell must exist and be executable.

system Whether the user is a system user with lower UID. Valid values are true, false.

uid The user ID. Must be specified numerically. For new users being created, if no user ID is specified then one will be chosen automatically, which will likely result in the same user having different IDs on different systems, which is not recommended. This is especially noteworthy if you use Puppet to manage the same user on both Darwin and other platforms, since Puppet does the ID generation for you on Darwin, but the tools do so on other platforms.

---

vcsrepo

A local version control repository

Features

- *bare_repositories*: The provider differentiates between bare repositories and those with working copies
- *filesystem_types*: The provider supports different filesystem types
- *gzip_compression*: The provider supports explicit GZip compression levels
- *reference_tracking*: The provider supports tracking revision references that can change over time (eg, some VCS tags and branch names)

Provider bare_repositories filesystem_types gzip_compression reference_tracking bzr *X* cvs *X* *X* git *X* *X* hg *X* svn *X* *X*

Parameters

compression Compression level Requires features gzip_compression.

ensure Valid values are present, bare, absent, latest.

fstype Filesystem type Requires features filesystem_types.

path

- **namevar**

Absolute path to repository

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **bzr**: Supports Bazaar repositories Required binaries: bzr. Default for bzr == exists. Supported features: reference_tracking.
- **cvs**: Supports CVS repositories/workspaces Required binaries: cvs. Default for cvs == exists. Supported features: gzip_compression, reference_tracking.
- **git**: Supports Git repositories Required binaries: git. Default for git == exists. Supported features: bare_repositories, reference_tracking.
- **hg**: Supports Mercurial repositories Required binaries: hg. Default for hg == exists. Supported features: reference_tracking.
- **svn**: Supports Subversion repositories Required binaries: svn, svnadmin. Default for svn == exists. Supported features: filesystem_types, reference_tracking.

revision The revision of the repository Values can match /^\S+$/.

source The source URI for the repository

---

vlan

This represents a router or switch vlan.

Parameters

description Vlan name

device_url Url to connect to a router or switch.

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

name Vlan id. It must be a number Values can match /^\d+/.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **cisco**: Cisco switch/router provider for vlans.

---

yumrepo

The client-side description of a yum repository. Repository configurations are found by parsing /etc/yum.conf and the files indicated by the reposdir option in that file (see yum.conf(5) for details)

Most parameters are identical to the ones documented in yum.conf(5)

Continuation lines that yum supports for example for the baseurl are not supported. No attempt is made to access files included with the **include** directive

Parameters

baseurl The URL for this repository. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

cost Cost of this repository. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /\d+/.

descr A human readable description of the repository. This corresponds to the name parameter in yum.conf(5). Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

enabled Whether this repository is enabled or disabled. Possible values are '0', and '1'. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /(0|1)/.

enablegroups Determines whether yum will allow the use of package groups for this repository. Possible values are '0', and '1'. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /(0|1)/.

exclude List of shell globs. Matching packages will never be considered in updates or installs for this repo. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

failovermethod Either 'roundrobin' or 'priority'. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /roundrobin|priority/.

gpgcheck Whether to check the GPG signature on packages installed from this repository. Possible values are '0', and '1'.

Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /(0|1)/.

gpgkey The URL for the GPG key with which packages from this repository are signed. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

http_caching Either 'packages' or 'all' or 'none'. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /packages|all|none/.

include A URL from which to include the config. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

includepkgs List of shell globs. If this is set, only packages matching one of the globs will be considered for update or install. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

keepalive Either '1' or '0'. This tells yum whether or not HTTP/1.1 keepalive should be used with this repository. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /(0|1)/.

metadata_expire Number of seconds after which the metadata will expire. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /[0−9]+/.

mirrorlist The URL that holds the list of mirrors for this repository. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

name The name of the repository. This corresponds to the repositoryid parameter in yum.conf(5).

priority Priority of this repository from 1–99. Requires that the priorities plugin is installed and enabled. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /[1−9][0−9]?/.

protect Enable or disable protection for this repository. Requires that the protectbase plugin is installed and enabled. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /(0|1)/.

proxy URL to the proxy server for this repository. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

proxy_password Password for this proxy. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

proxy_username Username for this proxy. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /.*/.

timeout Number of seconds to wait for a connection before timing out. Set this to 'absent' to remove it from the file completely Valid values are absent. Values can match /[0−9]+/.

---

zfs

Manage zfs. Create destroy and set properties on zfs instances.

Autorequires: If Puppet is managing the zpool at the root of this zfs instance, the zfs resource will autorequire it. If Puppet is managing any parent zfs instances, the zfs resource will autorequire them.

Parameters

aclinherit The aclinherit property. Values: discard noallow restricted passthrough passthrough-x

aclmode The aclmode property. Values: discard groupmask passthrough

atime The atime property. Values: on off

canmount The canmount property. Values: on off noauto

checksum The checksum property. Values: on off fletcher2 fletcher4 sha256

compression The compression property. Values: on off lzjb gzip gzip-[1–9] zle

copies The copies property. Values: 1 2 3

devices The devices property. Values: on off

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

exec The exec property. Values: on off

logbias The logbias property. Values: latency throughput

mountpoint The mountpoint property. Values: legacy none

name The full name for this filesystem. (including the zpool)

nbmand The nbmand property. Values: on off

primarycache The primarycache property. Values: all none metadata

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **solaris**: Provider for Solaris zfs. Required binaries: /usr/sbin/zfs. Default for operatingsystem == solaris.

quota The quota property. Values: none

readonly The readonly property. Values: on off

recordsize The recordsize property. Values: 512 to 128k, power of 2

refquota The refquota property. Values: none

refreservation The refreservation property. Values: none

reservation The reservation property. Values: none

secondarycache The secondarycache property. Values: all none metadata

setuid The setuid property. Values: on off

shareiscsi The shareiscsi property. Values: on off type=

sharenfs The sharenfs property. Values: on off share(1M) options

sharesmb The sharesmb property. Values: on off sharemgr(1M) options

snapdir The snapdir property. Values: hidden visible

version The version property. Values: 1 2 3 4 current

volsize The volsize property. Values:

vscan The vscan property. Values: on off

xattr The xattr property. Values: on off

zoned The zoned property. Values: on off

---

zone

Solaris zones.

Autorequires: If Puppet is managing the directory specified as the root of the zone's filesystem (with the path attribute), the zone resource will autorequire that directory.

Parameters

autoboot Whether the zone should automatically boot. Valid values are true, false.

clone Instead of installing the zone, clone it from another zone. If the zone root resides on a zfs file system, a snapshot will be used to create the clone, is it redisides on ufs, a copy of the zone will be used. The zone you clone from must not be running.

create_args Arguments to the zonecfg create command. This can be used to create branded zones.

dataset The list of datasets delegated to the non global zone from the global zone. All datasets must be zfs filesystem names which is different than the mountpoint.

ensure The running state of the zone. The valid states directly reflect the states that zoneadm provides. The states are linear, in that a zone must be configured then installed, and only then can be running. Note also that halt is currently used to stop zones.

id The numerical ID of the zone. This number is autogenerated and cannot be changed.

inherit The list of directories that the zone inherits from the global zone. All directories must be fully qualified.

install_args Arguments to the zoneadm install command. This can be used to create branded zones.

ip The IP address of the zone. IP addresses must be specified with the interface, separated by a colon, e.g.: bge0:192.168.0.1. For multiple interfaces, specify them in an array.

iptype The IP stack type of the zone. Can either be 'shared' or 'exclusive'. Valid values are shared, exclusive.

name The name of the zone.

path The root of the zone's filesystem. Must be a fully qualified file name. If you include '%s' in the path, then it will be replaced with the zone's name. At this point, you cannot use Puppet to move a zone.

pool The resource pool for this zone.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **solaris**: Provider for Solaris Zones. Required binaries: /usr/sbin/zoneadm, /usr/sbin/zonecfg. Default for operatingsystem == solaris.

realhostname The actual hostname of the zone.

shares Number of FSS CPU shares allocated to the zone.

sysidcfg The text to go into the sysidcfg file when the zone is first booted. The best way is to use a template:

```
# $templatedir/sysidcfg
system_locale=en_US
timezone=GMT
terminal=xterms
security_policy=NONE
root_password=<%= password %>
timeserver=localhost
name_service=DNS {domain_name=<%= domain %> name_server=<%= nameserver %>}
network_interface=primary {hostname=<%= realhostname %>
  ip_address=<%= ip %>
  netmask=<%= netmask %>
  protocol_ipv6=no
  default_route=<%= defaultroute %>}
nfs4_domain=dynamic
```

And then call that:

```
zone { myzone:
  ip => "bge0:192.168.0.23",
  sysidcfg => template(sysidcfg),
  path => "/opt/zones/myzone",
  realhostname => "fully.qualified.domain.name"
}
```

The sysidcfg only matters on the first booting of the zone, so Puppet only checks for it at that time.

---

zpool

Manage zpools. Create and delete zpools. The provider WILL NOT SYNC, only report differences.

Supports vdevs with mirrors, raidz, logs and spares.

Parameters

disk The disk(s) for this pool. Can be an array or space separated string

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

log Log disks for this pool. (doesn't support mirroring yet)

mirror List of all the devices to mirror for this pool. Each mirror should be a space separated string:

```
mirror => ["disk1 disk2", "disk3 disk4"],
```

pool

- **namevar**

The name for this pool.

provider The specific backend for provider to use. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

- **solaris**: Provider for Solaris zpool. Required binaries: /usr/sbin/zpool. Default for operatingsystem == solaris.

raid_parity Determines parity when using raidz property.

raidz List of all the devices to raid for this pool. Should be an array of space separated strings:

```
raidz => ["disk1 disk2", "disk3 disk4"],
```

spare Spare disk(s) for this pool.

This page autogenerated on Sat May 21 17:20:22 –0700 2011

Chapter 58

Function Reference

This page is autogenerated; any changes will get overwritten *(last generated on Sat May 21 17:19:37 –0700 2011)*

There are two types of functions in Puppet: Statements and rvalues. Statements stand on their own and do not return arguments; they are used for performing stand-alone work like importing. Rvalues return values and can only be used in a statement requiring a value, such as an assignment or a case statement.

Functions execute on the Puppet master. They do not execute on the Puppet agent.
Hence they only have access to the commands and data available on the Puppet master host.

Here are the functions available in Puppet:

alert

Log a message on the server at level alert.

- *Type*: statement

create_resources

Converts a hash into a set of resources and adds them to the catalog. Takes two parameters: create_resource($type, $resources) Creates resources of type $type from the $resources hash. Assumes that hash is in the following form: {title=>{parameters}} This is currently tested for defined resources, classes, as well as native types

- *Type*: statement

crit

Log a message on the server at level crit.

- *Type*: statement

debug

Log a message on the server at level debug.

- *Type*: statement

defined

Determine whether a given class or resource type is defined. This function can also determine whether a specific resource has been declared. Returns true or false. Accepts class names, type names, and resource references.

The defined function checks both native and defined types, including types provided as plugins via modules. Types and classes are both checked using their names:

```
defined("file")
defined("customtype")
defined("foo")
defined("foo::bar")
```

Resource declarations are checked using resource references, e.g. defined(File['/tmp/myfile']). Checking whether a given resource has been declared is, unfortunately, dependent on the parse order of the configuration, and the following code will not work:

```
if defined(File['/tmp/foo']) {
    notify("This configuration includes the /tmp/foo file.")
}
file {"/tmp/foo":
    ensure => present,
}
```

However, this order requirement refers to parse order only, and ordering of resources in the configuration graph (e.g. with before or require) does not affect the behavior of defined.

- *Type*: rvalue

emerg

Log a message on the server at level emerg.

- *Type*: statement

err

Log a message on the server at level err.

- *Type*: statement

extlookup

This is a parser function to read data from external files, this version uses CSV files but the concept can easily be adjust for databases, yaml or any other queryable data source.

The object of this is to make it obvious when it's being used, rather than magically loading data in when an module is loaded I prefer to look at the code and see statements like:

```
$snmp_contact = extlookup("snmp_contact")
```

The above snippet will load the snmp_contact value from CSV files, this in its own is useful but a common construct in puppet manifests is something like this:

```
case $domain {
  "myclient.com": { $snmp_contact = "John Doe <john@myclient.com>" }
  default:        { $snmp_contact = "My Support <support@my.com>" }
}
```

Over time there will be a lot of this kind of thing spread all over your manifests and adding an additional client involves grepping through manifests to find all the places where you have constructs like this.

This is a data problem and shouldn't be handled in code, a using this function you can do just that.

First you configure it in site.pp:

```
$extlookup_datadir = "/etc/puppet/manifests/extdata"
$extlookup_precedence = ["%{fqdn}", "domain_%{domain}", "common"]
```

The array tells the code how to resolve values, first it will try to find it in web1.myclient.com.csv then in domain_myclient.com.csv and finally in common.csv

Now create the following data files in /etc/puppet/manifests/extdata:

```
domain_myclient.com.csv:
  snmp_contact,John Doe <john@myclient.com>
  root_contact,support@%{domain}
  client_trusted_ips,192.168.1.130,192.168.10.0/24

common.csv:
  snmp_contact,My Support <support@my.com>
  root_contact,support@my.com
```

Now you can replace the case statement with the simple single line to achieve the exact same outcome:

$snmp_contact = extlookup("snmp_contact")

The above code shows some other features, you can use any fact or variable that is in scope by simply using %{varname} in your data files, you can return arrays by just having multiple values in the csv after the initial variable name.

In the event that a variable is nowhere to be found a critical error will be raised that will prevent your manifest from compiling, this is to avoid accidentally putting in empty values etc. You can however specify a default value:

$ntp_servers = extlookup("ntp_servers", "1.${country}.pool.ntp.org")

In this case it will default to "1.${country}.pool.ntp.org" if nothing is defined in any data file.

You can also specify an additional data file to search first before any others at use time, for example:

```
$version = extlookup("rsyslog_version", "present", "packages")
package{"rsyslog": ensure => $version }
```

This will look for a version configured in packages.csv and then in the rest as configured by $extlookup_precedence if it's not found anywhere it will default to present, this kind of use case makes puppet a lot nicer for managing large amounts of packages since you do not need to edit a load of manifests to do simple things like adjust a desired version number.

Precedence values can have variables embedded in them in the form %{fqdn}, you could for example do:

```
$extlookup_precedence = ["hosts/%{fqdn}", "common"]
```

This will result in /path/to/extdata/hosts/your.box.com.csv being searched.

This is for back compatibility to interpolate variables with %. % interpolation is a workaround for a problem that has been fixed: Puppet variable interpolation at top scope used to only happen on each run.

- *Type*: rvalue

fail

Fail with a parse error.

- *Type*: statement

file

Return the contents of a file. Multiple files can be passed, and the first file that exists will be read in.

- *Type*: rvalue

fqdn_rand

Generates random numbers based on the node's fqdn. Generated random values will be a range from 0 up to and excluding n, where n is the first parameter. The second argument specifies a number to add to the seed and is optional, for example:

```
$random_number = fqdn_rand(30)
$random_number_seed = fqdn_rand(30,30)
```

- *Type*: rvalue

generate

Calls an external command on the Puppet master and returns the results of the command. Any arguments are passed to the external command as arguments. If the generator does not exit with return code of 0, the generator is considered to have failed and a parse error is thrown. Generators can only have file separators, alphanumerics, dashes, and periods in them. This function will attempt to protect you from malicious generator calls (e.g., those with '..' in them), but it can never be entirely safe. No subshell is used to execute generators, so all shell metacharacters are passed directly to the generator.

- *Type*: rvalue

include

Evaluate one or more classes.

- *Type*: statement

info

Log a message on the server at level info.

- *Type*: statement

inline_template

Evaluate a template string and return its value. See the templating docs for more information. Note that if multiple template strings are specified, their output is all concatenated and returned as the output of the function.

- *Type*: rvalue

md5

Returns a MD5 hash value from a provided string.

- *Type*: rvalue

notice

Log a message on the server at level notice.

- *Type*: statement

realize

Make a virtual object real. This is useful when you want to know the name of the virtual object and don't want to bother with a full collection. It is slightly faster than a collection, and, of course, is a bit shorter. You must pass the object using a reference; e.g.: realize User[luke].

- *Type*: statement

regsubst

Perform regexp replacement on a string or array of strings.

- *Parameters* (in order):
  - *target* The string or array of strings to operate on. If an array, the replacement will be performed on each of the elements in the array, and the return value will be an array.
  - *regexp* The regular expression matching the target string. If you want it anchored at the start and or end of the string, you must do that with ^ and $ yourself.
  - *replacement* Replacement string. Can contain backreferences to what was matched using \0 (whole match), \1 (first set of parentheses), and so on.
  - *flags* Optional. String of single letter flags for how the regexp is interpreted:
    * *E* Extended regexps
    * *I* Ignore case in regexps
    * *M* Multiline regexps
    * *G* Global replacement; all occurrences of the regexp in each target string will be replaced. Without this, only the first occurrence will be replaced.
  - *encoding* Optional. How to handle multibyte characters. A single-character string with the following values:
    * *N* None
    * *E* EUC
    * *S* SJIS
    * *U* UTF–8
- *Examples*

Get the third octet from the node's IP address:

```
$i3 = regsubst($ipaddress,'^(\d+)\.(\d+)\.(\d+)\.(\d+)$','\3')
```

Put angle brackets around each octet in the node's IP address:

```
$x = regsubst($ipaddress, '([0-9]+)', '<\1>', 'G')
```

- *Type*: rvalue

require

Evaluate one or more classes, adding the required class as a dependency.

The relationship metaparameters work well for specifying relationships between individual resources, but they can be clumsy for specifying relationships between classes. This function is a superset of the 'include' function, adding a class relationship so that the requiring class depends on the required class.

Warning: using require in place of include can lead to unwanted dependency cycles.

For instance the following manifest, with 'require' instead of 'include' would produce a nasty dependence cycle, because notify imposes a before between File[/foo] and Service[foo]:

```
class myservice {
  service { foo: ensure => running }
}

class otherstuff {
  include myservice
  file { '/foo': notify => Service[foo] }
}
```

Note that this function only works with clients 0.25 and later, and it will fail if used with earlier clients.

- *Type*: statement

search

Add another namespace for this class to search. This allows you to create classes with sets of definitions and add those classes to another class's search path.

- *Type*: statement

sha1

Returns a SHA1 hash value from a provided string.

- *Type*: rvalue

shellquote

Quote and concatenate arguments for use in Bourne shell.

Each argument is quoted separately, and then all are concatenated with spaces. If an argument is an array, the elements of that array is interpolated within the rest of the arguments; this makes it possible to have an array of arguments and pass that array to shellquote instead of having to specify each argument individually in the call.

- *Type*: rvalue

split

Split a string variable into an array using the specified split regexp.

Example:

```
$string     = 'v1.v2:v3.v4'
$array_var1 = split($string, ':')
$array_var2 = split($string, '[.]')
$array_var3 = split($string, '[.:]')
```

$array_var1 now holds the result ['v1.v2', 'v3.v4'], while $array_var2 holds ['v1', 'v2:v3', 'v4'], and $array_var3 holds ['v1', 'v2', 'v3', 'v4'].

Note that in the second example, we split on a literal string that contains a regexp meta-character (.), which must be escaped. A simple way to do that for a single character is to enclose it in square brackets; a backslash will also escape a single character.

- *Type*: rvalue

sprintf

Perform printf-style formatting of text.

The first parameter is format string describing how the rest of the parameters should be formatted. See the documentation for the Kernel::sprintf function in Ruby for all the details.

- *Type*: rvalue

tag

Add the specified tags to the containing class or definition. All contained objects will then acquire that tag, also.

- *Type*: statement

tagged

A boolean function that tells you whether the current container is tagged with the specified tags. The tags are ANDed, so that all of the specified tags must be included for the function to return true.

- *Type*: rvalue

template

Evaluate a template and return its value. See the templating docs for more information.

Note that if multiple templates are specified, their output is all concatenated and returned as the output of the function.

- *Type*: rvalue

versioncmp

Compares two versions

Prototype:

```
$result = versioncmp(a, b)
```

Where a and b are arbitrary version strings

This functions returns a number:

- Greater than 0 if version a is greater than version b
- Equal to 0 if both version are equals

- Less than 0 if version a is less than version b

Example:

```
if versioncmp('2.6-1', '2.4.5') > 0 {
    notice('2.6-1 is > than 2.4.5')
}
```

- *Type*: rvalue

warning

Log a message on the server at level warning.

- *Type*: statement

This page autogenerated on Sat May 21 17:19:37 -0700 2011

Chapter 59

Metaparameter Reference

This page is autogenerated; any changes will get overwritten *(last generated on Sat May 21 17:19:52 –0700 2011)*

Chapter 60

Metaparameters

Metaparameters are parameters that work with any resource type; they are part of the Puppet framework itself rather than being part of the implementation of any given instance. Thus, any defined metaparameter can be used with any instance in your manifest, including defined components.

Available Metaparameters

alias

Creates an alias for the object. Puppet uses this internally when you provide a symbolic title:

```
file { 'sshdconfig':
  path => $operatingsystem ? {
    solaris => "/usr/local/etc/ssh/sshd_config",
    default => "/etc/ssh/sshd_config"
  },
  source => "..."
}

service { 'sshd':
  subscribe => File['sshdconfig']
}
```

When you use this feature, the parser sets sshdconfig as the title, and the library sets that as an alias for the file so the dependency lookup in Service['sshd'] works. You can use this metaparameter yourself, but note that only the library can use these aliases; for instance, the following code will not work:

```
file { "/etc/ssh/sshd_config":
  owner => root,
  group => root,
  alias => 'sshdconfig'
}

file { 'sshdconfig':
  mode => 644
}
```

There's no way here for the Puppet parser to know that these two stanzas should be affecting the same file.

See the Language Guide for more information.

audit

Marks a subset of this resource's unmanaged attributes for auditing. Accepts an attribute name or a list of attribute names.

Auditing a resource attribute has two effects: First, whenever a catalog is applied with puppet apply or puppet agent, Puppet will check whether that attribute of the resource has been modified, comparing its current value

to the previous run; any change will be logged alongside any actions performed by Puppet while applying the catalog.

Secondly, marking a resource attribute for auditing will include that attribute in inspection reports generated by puppet inspect; see the puppet inspect documentation for more details.

Managed attributes for a resource can also be audited, but note that changes made by Puppet will be logged as additional modifications. (I.e. if a user manually edits a file whose contents are audited and managed, puppet agent's next two runs will both log an audit notice: the first run will log the user's edit and then revert the file to the desired state, and the second run will log the edit made by Puppet.)

before

References to one or more objects that depend on this object. This parameter is the opposite of **require** — it guarantees that the specified object is applied later than the specifying object:

```
file { "/var/nagios/configuration":
  source  => "...",
  recurse => true,
  before  => Exec["nagios-rebuid"]
}

exec { "nagios-rebuild":
  command => "/usr/bin/make",
  cwd     => "/var/nagios/configuration"
}
```

This will make sure all of the files are up to date before the make command is run.

check

Audit specified attributes of resources over time, and report if any have changed. This parameter has been deprecated in favor of 'audit'.

loglevel

Sets the level that information will be logged. The log levels have the biggest impact when logs are sent to syslog (which is currently the default). Valid values are debug, info, notice, warning, err, alert, emerg, crit, verbose.

noop

Boolean flag indicating whether work should actually be done. Valid values are true, false.

notify

References to one or more objects that depend on this object. This parameter is the opposite of **subscribe** — it creates a dependency relationship like **before,** and also causes the dependent object(s) to be refreshed when this object is changed. For instance:

```
file { "/etc/sshd_config":
  source => "....",
  notify => Service['sshd']
}

service { 'sshd':
  ensure => running
}
```

This will restart the sshd service if the sshd config file changes.

require

References to one or more objects that this object depends on. This is used purely for guaranteeing that changes to required objects happen before the dependent object. For instance:

```
# Create the destination directory before you copy things down
file { "/usr/local/scripts":
  ensure => directory
}

file { "/usr/local/scripts/myscript":
  source  => "puppet://server/module/myscript",
  mode    => 755,
  require => File["/usr/local/scripts"]
}
```

Multiple dependencies can be specified by providing a comma-seperated list of resources, enclosed in square brackets:

```
require => [ File["/usr/local"], File["/usr/local/scripts"] ]
```

Note that Puppet will autorequire everything that it can, and there are hooks in place so that it's easy for resources to add new ways to autorequire objects, so if you think Puppet could be smarter here, let us know.

In fact, the above code was redundant — Puppet will autorequire any parent directories that are being managed; it will automatically realize that the parent directory should be created before the script is pulled down.

Currently, exec resources will autorequire their CWD (if it is specified) plus any fully qualified paths that appear in the command. For instance, if you had an exec command that ran the myscript mentioned above, the above code that pulls the file down would be automatically listed as a requirement to the exec code, so that you would always be running againts the most recent version.

schedule

On what schedule the object should be managed. You must create a schedule object, and then reference the name of that object to use that for your schedule:

```
schedule { 'daily':
  period => daily,
  range  => "2-4"
}

exec { "/usr/bin/apt-get update":
  schedule => 'daily'
}
```

The creation of the schedule object does not need to appear in the configuration before objects that use it.

stage

Which run stage a given resource should reside in. This just creates a dependency on or from the named milestone. For instance, saying that this is in the 'bootstrap' stage creates a dependency on the 'bootstrap' milestone.

By default, all classes get directly added to the 'main' stage. You can create new stages as resources:

```
stage { ['pre', 'post']: }
```

To order stages, use standard relationships:

```
stage { 'pre': before => Stage['main'] }
```

Or use the new relationship syntax:

```
Stage['pre'] -> Stage['main'] -> Stage['post']
```

Then use the new class parameters to specify a stage:

```
class { 'foo': stage => 'pre' }
```

Stages can only be set on classes, not individual resources. This will fail:

```
file { '/foo': stage => 'pre', ensure => file }
```

subscribe

References to one or more objects that this object depends on. This metaparameter creates a dependency relationship like **require,** and also causes the dependent object to be refreshed when the subscribed object is changed. For instance:

```
class nagios {
  file { 'nagconf':
    path   => "/etc/nagios/nagios.conf"
    source => "puppet://server/module/nagios.conf",
  }
  service { 'nagios':
    ensure    => running,
    subscribe => File['nagconf']
  }
}
```

Currently the exec, mount and service types support refreshing.

tag

Add the specified tags to the associated resource. While all resources are automatically tagged with as much information as possible (e.g., each class and definition containing the resource), it can be useful to add your own tags to a given resource.

Tags are currently useful for things like applying a subset of a host's configuration:

```
puppet agent --test --tags mytag
```

This way, when you're testing a configuration you can run just the portion you're testing.

This page autogenerated on Sat May 21 17:19:59 -0700 2011

Chapter 61

Configuration Reference

This page is autogenerated; any changes will get overwritten *(last generated on Sat May 21 17:19:30 -0700 2011)*

Specifying Configuration Parameters

On The Command-Line

Every Puppet executable (with the exception of puppetdoc) accepts all of the parameters below, but not all of the arguments make sense for every executable.

I have tried to be as thorough as possible in the descriptions of the arguments, so it should be obvious whether an argument is appropriate or not.

These parameters can be supplied to the executables either as command-line options or in the configuration file. For instance, the command-line invocation below would set the configuration directory to /private/puppet:

```
$ puppet agent --confdir=/private/puppet
```

Note that boolean options are turned on and off with a slightly different syntax on the command line:

```
$ puppet agent --storeconfigs
```

```
$ puppet agent --no-storeconfigs
```

The invocations above will enable and disable, respectively, the storage of the client configuration.

Configuration Files

As mentioned above, the configuration parameters can also be stored in a configuration file, located in the configuration directory. As root, the default configuration directory is /etc/puppet, and as a regular user, the default configuration directory is ~user/.puppet. As of 0.23.0, all executables look for puppet.conf in their configuration directory (although they previously looked for separate files). For example, puppet.conf is located at /etc/puppet/puppet.conf as root and ~user/.puppet/puppet.conf as a regular user by default.

All executables will set any parameters set within the [main] section, and each executable will also use one of the [master], [agent].

File Format

The file follows INI-style formatting. Here is an example of a very simple puppet.conf file:

```
[main]
  confdir = /private/puppet
  storeconfigs = true
```

Note that boolean parameters must be explicitly specified as true or false as seen above.

If you need to change file or directory parameters (e.g., reset the mode or owner), do so within curly braces on the same line:

```
[main]
  vardir = /new/vardir {owner = root, mode = 644}
```

If you're starting out with a fresh configuration, you may wish to let the executable generate a template configuration file for you by invoking the executable in question with the --genconfig command. The executable will print a template configuration to standard output, which can be redirected to a file like so:

```
$ puppet agent --genconfig > /etc/puppet/puppet.conf
```

Note that this invocation will replace the contents of any pre-existing puppet.conf file, so make a backup of your present config if it contains valuable information.

Like the --genconfig argument, the executables also accept a --genmanifest argument, which will generate a manifest that can be used to manage all of Puppet's directories and files and prints it to standard output. This can likewise be redirected to a file:

```
$ puppet agent --genmanifest > /etc/puppet/manifests/site.pp
```

Puppet can also create user and group accounts for itself (one puppet group and one puppet user) if it is invoked as root with the --mkusers argument:

```
$ puppet master --mkusers
```

Signals

The puppet agent and puppet master executables catch some signals for special handling. Both daemons catch (SIGHUP), which forces the server to restart tself. Predictably, interrupt and terminate (SIGINT and SIGTERM) will shut down the server, whether it be an instance of puppet agent or puppet master.

Sending the SIGUSR1 signal to an instance of puppet agent will cause it to immediately begin a new configuration transaction with the server. This signal has no effect on puppet master.

Configuration Parameter Reference

Below is a list of all documented parameters. Not all of them are valid with all Puppet executables, but the executables will ignore any inappropriate values.

allow_duplicate_certs

Whether to allow a new certificate request to overwrite an existing certificate.

- *Default*: false

archive_file_server

During an inspect run, the file bucket server to archive files to if archive_files is set.

- *Default*: $server

archive_files

During an inspect run, whether to archive files whose contents are audited to a file bucket.

- *Default*: false

async_storeconfigs

Whether to use a queueing system to provide asynchronous database integration. Requires that puppetqd be running and that 'PSON' support for ruby be installed.

- *Default*: false

authconfig

The configuration file that defines the rights to the different namespaces and methods. This can be used as a coarse-grained authorization system for both puppet agent and puppet master.

- *Default*: $confdir/namespaceauth.conf

autoflush

Whether log files should always flush to disk.

- *Default*: false

autosign

Whether to enable autosign. Valid values are true (which autosigns any key request, and is a very bad idea), false (which never autosigns any key request), and the path to a file, which uses that configuration file to determine which keys to sign.

- *Default*: $confdir/autosign.conf

bindaddress

The address a listening server should bind to. Mongrel servers default to 127.0.0.1 and WEBrick defaults to 0.0.0.0.

bucketdir

Where FileBucket files are stored.

- *Default*: $vardir/bucket

ca

Wether the master should function as a certificate authority.

- *Default*: true

ca_days

How long a certificate should be valid. This parameter is deprecated, use ca_ttl instead

ca_md

The type of hash used in certificates.

- *Default*: md5

ca_name

The name to use the Certificate Authority certificate.

- *Default*: Puppet CA: $certname

ca_port

The port to use for the certificate authority.

- *Default*: $masterport

ca_server

The server to use for certificate authority requests. It's a separate server because it cannot and does not need to horizontally scale.

- *Default*: $server

ca_ttl

The default TTL for new certificates; valid values must be an integer, optionally followed by one of the units 'y' (years of 365 days), 'd' (days), 'h' (hours), or 's' (seconds). The unit defaults to seconds. If this parameter is set, ca_days is ignored. Examples are '3600' (one hour) and '1825d', which is the same as '5y' (5 years)

- *Default*: 5y

cacert

The CA certificate.

- *Default*: $cadir/ca_crt.pem

cacrl

The certificate revocation list (CRL) for the CA. Will be used if present but otherwise ignored.

- *Default*: $cadir/ca_crl.pem

cadir

The root directory for the certificate authority.

- *Default*: $ssldir/ca

cakey

The CA private key.

- *Default*: $cadir/ca_key.pem

capass

Where the CA stores the password for the private key

- *Default*: $caprivatedir/ca.pass

caprivatedir

Where the CA stores private certificate information.

- *Default*: $cadir/private

capub

The CA public key.

- *Default*: $cadir/ca_pub.pem

catalog_format

(Deprecated for 'preferred_serialization_format') What format to use to dump the catalog. Only supports 'marshal' and 'yaml'. Only matters on the client, since it asks the server for a specific format.

catalog_terminus

Where to get node catalogs. This is useful to change if, for instance, you'd like to pre-compile catalogs and store them in memcached or some other easily-accessed store.

- *Default*: compiler

cert_inventory

A Complete listing of all certificates

- *Default*: $cadir/inventory.txt

certdir

The certificate directory.

- *Default*: $ssldir/certs

certdnsnames

The DNS names on the Server certificate as a colon-separated list. If it's anything other than an empty string, it will be used as an alias in the created certificate. By default, only the server gets an alias set up, and only for 'puppet'.

certificate_revocation

Whether certificate revocation should be supported by downloading a Certificate Revocation List (CRL) to all clients. If enabled, CA chaining will almost definitely not work.

- *Default*: true

certname

The name to use when handling certificates. Defaults to the fully qualified domain name.

- *Default*: pelin.lovedthanlost.net

classfile

The file in which puppet agent stores a list of the classes associated with the retrieved configuration. Can be loaded in the separate puppet executable using the ––loadclasses option.

- *Default*: $statedir/classes.txt

client_datadir

The directory in which serialized data is stored on the client.

- *Default*: $vardir/client_data

clientbucketdir

Where FileBucket files are stored locally.

- *Default*: $vardir/clientbucket

clientyamldir

The directory in which client-side YAML data is stored.

- *Default*: $vardir/client_yaml

code

Code to parse directly. This is essentially only used by puppet, and should only be set if you're writing your own Puppet executable

color

Whether to use colors when logging to the console. Valid values are ansi (equivalent to true), html (mostly used during testing with TextMate), and false, which produces no color.

- *Default*: ansi

confdir

The main Puppet configuration directory. The default for this parameter is calculated based on the user. If the process is running as root or the user that Puppet is supposed to run as, it defaults to a system directory, but if it's running as any other user, it defaults to being in the user's home directory.

- *Default*: /etc/puppet

config

The configuration file for doc.

- *Default*: $confdir/puppet.conf

config_version

How to determine the configuration version. By default, it will be the time that the configuration is parsed, but you can provide a shell script to override how the version is determined. The output of this script will be added to every log message in the reports, allowing you to correlate changes on your hosts to the source version on the server.

configprint

Print the value of a specific configuration parameter. If a parameter is provided for this, then the value is printed and puppet exits. Comma-separate multiple values. For a list of all values, specify 'all'. This feature is only available in Puppet versions higher than 0.18.4.

configtimeout

How long the client should wait for the configuration to be retrieved before considering it a failure. This can help reduce flapping if too many clients contact the server at one time.

- *Default*: 120

couchdb_url

The url where the puppet couchdb database will be created

- *Default*: http://127.0.0.1:5984/puppet

csrdir

Where the CA stores certificate requests

- *Default*: $cadir/requests

daemonize

Send the process into the background. This is the default.

- *Default*: true

dbadapter

The type of database to use.

- *Default*: sqlite3

dbconnections

The number of database connections for networked databases. Will be ignored unless the value is a positive integer.

dblocation

The database cache for client configurations. Used for querying within the language.

- *Default*: $statedir/clientconfigs.sqlite3

dbmigrate

Whether to automatically migrate the database.

- *Default*: false

dbname

The name of the database to use.

- *Default*: puppet

dbpassword

The database password for caching. Only used when networked databases are used.

- *Default*: puppet

dbport

The database password for caching. Only used when networked databases are used.

dbserver

The database server for caching. Only used when networked databases are used.

- *Default*: localhost

dbsocket

The database socket location. Only used when networked databases are used. Will be ignored if the value is an empty string.

dbuser

The database user for caching. Only used when networked databases are used.

- *Default*: puppet

deviceconfig

Path to the device config file for puppet device

- *Default*: $confdir/device.conf

devicedir

The root directory of devices' $vardir

- *Default*: $vardir/devices

diff

Which diff command to use when printing differences between files.

- *Default*: diff

diff_args

Which arguments to pass to the diff command when printing differences between files.

- *Default*: -u

document_all

Document all resources

- *Default*: false

downcasefacts

Whether facts should be made all lowercase when sent to the server.

- *Default*: false

dynamicfacts

Facts that are dynamic; these facts will be ignored when deciding whether changed facts should result in a recompile. Multiple facts should be comma-separated.

- *Default*: memorysize,memoryfree,swapsize,swapfree

environment

The environment Puppet is running in. For clients (e.g., puppet agent) this determines the environment itself, which is used to find modules and much more. For servers (i.e., puppet master) this provides the default environment for nodes we know nothing about.

- *Default*: production

evaltrace

Whether each resource should log when it is being evaluated. This allows you to interactively see exactly what is being done.

- *Default*: false

external_nodes

An external command that can produce node information. The output must be a YAML dump of a hash, and that hash must have one or both of classes and parameters, where classes is an array and parameters is a hash. For unknown nodes, the commands should exit with a non-zero exit code. This command makes it straightforward to store your node mapping information in other data sources like databases.

- *Default*: none

factdest

Where Puppet should store facts that it pulls down from the central server.

- *Default*: $vardir/facts/

factpath

Where Puppet should look for facts. Multiple directories should be colon-separated, like normal PATH variables.

- *Default*: $vardir/lib/facter:$vardir/facts

facts_terminus

The node facts terminus.

- *Default*: facter

factsignore

What files to ignore when pulling down facts.

- *Default*: .svn CVS

factsource

From where to retrieve facts. The standard Puppet file type is used for retrieval, so anything that is a valid file source can be used here.

- *Default*: puppet://$server/facts/

factsync

Whether facts should be synced with the central server.

- *Default*: false

fileserverconfig

Where the fileserver configuration is stored.

- *Default*: $confdir/fileserver.conf

filetimeout

The minimum time to wait (in seconds) between checking for updates in configuration files. This timeout determines how quickly Puppet checks whether a file (such as manifests or templates) has changed on disk.

- *Default*: 15

freeze_main

Freezes the 'main' class, disallowing any code to be added to it. This essentially means that you can't have any code outside of a node, class, or definition other than in the site manifest.

- *Default*: false

genconfig

Whether to just print a configuration to stdout and exit. Only makes sense when used interactively. Takes into account arguments specified on the CLI.

- *Default*: false

genmanifest

Whether to just print a manifest to stdout and exit. Only makes sense when used interactively. Takes into account arguments specified on the CLI.

- *Default*: false

graph

Whether to create dot graph files for the different configuration graphs. These dot files can be interpreted by tools like OmniGraffle or dot (which is part of ImageMagick).

- *Default*: false

graphdir

Where to store dot-outputted graphs.

- *Default*: $statedir/graphs

group

The group puppet master should run as.

- *Default*: puppet

hostcert

Where individual hosts store and look for their certificates.

- *Default*: $certdir/$certname.pem

hostcrl

Where the host's certificate revocation list can be found. This is distinct from the certificate authority's CRL.

- *Default*: $ssldir/crl.pem

hostcsr

Where individual hosts store and look for their certificate requests.

- *Default*: $ssldir/csr_$certname.pem

hostprivkey

Where individual hosts store and look for their private key.

- *Default*: $privatekeydir/$certname.pem

hostpubkey

Where individual hosts store and look for their public key.

- *Default*: $publickeydir/$certname.pem

http_compression

Allow http compression in REST communication with the master. This setting might improve performance for agent -> master communications over slow WANs. Your puppet master needs to support compression (usually by activating some settings in a reverse-proxy in front of the puppet master, which rules out webrick). It is harmless to activate this settings if your master doesn't support compression, but if it supports it, this setting might reduce performance on high-speed LANs.

- *Default*: false

http_proxy_host

The HTTP proxy host to use for outgoing connections. Note: You may need to use a FQDN for the server hostname when using a proxy.

- *Default*: none

http_proxy_port

The HTTP proxy port to use for outgoing connections

- *Default*: 3128

httplog

Where the puppet agent web server logs.

- *Default*: $logdir/http.log

ignorecache

Ignore cache and always recompile the configuration. This is useful for testing new configurations, where the local cache may in fact be stale even if the timestamps are up to date - if the facts change or if the server changes.

- *Default*: false

ignoreimport

A parameter that can be used in commit hooks, since it enables you to parse-check a single file rather than requiring that all files exist.

- *Default*: false

ignoreschedules

Boolean; whether puppet agent should ignore schedules. This is useful for initial puppet agent runs.

- *Default*: false

inventory_port

The port to communicate with the inventory_server.

- *Default*: $masterport

inventory_server

The server to send facts to.

- *Default*: $server

inventory_terminus

Should usually be the same as the facts terminus

- *Default*: $facts_terminus

keylength

The bit length of keys.

- *Default*: 1024

lastrunfile

Where puppet agent stores the last run report summary in yaml format.

- *Default*: $statedir/last_run_summary.yaml

lastrunreport

Where puppet agent stores the last run report in yaml format.

- *Default*: $statedir/last_run_report.yaml

ldapattrs

The LDAP attributes to include when querying LDAP for nodes. All returned attributes are set as variables in the top-level scope. Multiple values should be comma-separated. The value 'all' returns all attributes.

- *Default*: all

ldapbase

The search base for LDAP searches. It's impossible to provide a meaningful default here, although the LDAP libraries might have one already set. Generally, it should be the 'ou=Hosts' branch under your main directory.

ldapclassattrs

The LDAP attributes to use to define Puppet classes. Values should be comma-separated.

- *Default*: puppetclass

ldapnodes

Whether to search for node configurations in LDAP. See http://projects.puppetlabs.com/projects/puppet/wiki/LDAP_
for more information.

- *Default*: false

ldapparentattr

The attribute to use to define the parent node.

- *Default*: parentnode

ldappassword

The password to use to connect to LDAP.

ldapport

The LDAP port. Only used if ldapnodes is enabled.

- *Default*: 389

ldapserver

The LDAP server. Only used if ldapnodes is enabled.

- *Default*: ldap

ldapssl

Whether SSL should be used when searching for nodes. Defaults to false because SSL usually requires certificates to be set up on the client side.

- *Default*: false

ldapstackedattrs

The LDAP attributes that should be stacked to arrays by adding the values in all hierarchy elements of the tree. Values should be comma-separated.

- *Default*: puppetvar

ldapstring

The search string used to find an LDAP node.

- *Default*: (&(objectclass=puppetClient)(cn=%s))

ldaptls

Whether TLS should be used when searching for nodes. Defaults to false because TLS usually requires certificates to be set up on the client side.

- *Default*: false

ldapuser

The user to use to connect to LDAP. Must be specified as a full DN.

lexical

Whether to use lexical scoping (vs. dynamic).

- *Default*: false

libdir

An extra search path for Puppet. This is only useful for those files that Puppet will load on demand, and is only guaranteed to work for those cases. In fact, the autoload mechanism is responsible for making sure this directory is in Ruby's search path

- *Default*: $vardir/lib

listen

Whether puppet agent should listen for connections. If this is true, then puppet agent will accept incoming REST API requests, subject to the default ACLs and the ACLs set in the rest_authconfig file. Puppet agent can respond usefully to requests on the run, facts, certificate, and resource endpoints.

- *Default*: false

localcacert

Where each client stores the CA certificate.

- *Default*: $certdir/ca.pem

localconfig

Where puppet agent caches the local configuration. An extension indicating the cache format is added automatically.

- *Default*: $statedir/localconfig

logdir

The Puppet log directory.

- *Default*: $vardir/log

manage_internal_file_permissions

Whether Puppet should manage the owner, group, and mode of files it uses internally

- *Default*: true

manifest

The entry-point manifest for puppet master.

- *Default*: $manifestdir/site.pp

manifestdir

Where puppet master looks for its manifests.

- *Default*: $confdir/manifests

masterhttplog

Where the puppet master web server logs.

- *Default*: $logdir/masterhttp.log

masterlog

Where puppet master logs. This is generally not used, since syslog is the default log destination.

- *Default*: $logdir/puppetmaster.log

masterport

Which port puppet master listens on.

- *Default*: 8140

maximum_uid

The maximum allowed UID. Some platforms use negative UIDs but then ship with tools that do not know how to handle signed ints, so the UIDs show up as huge numbers that can then not be fed back into the system. This is a hackish way to fail in a slightly more useful way when that happens.

- *Default*: 4294967290

mkusers

Whether to create the necessary user and group that puppet agent will run as.

- *Default*: false

modulepath

The search path for modules as a colon-separated list of directories.

- *Default*: $confdir/modules:/usr/share/puppet/modules

name

The name of the application, if we are running as one. The default is essentially $0 without the path or .rb.

- *Default*: doc

node_name

How the puppet master determines the client's identity and sets the 'hostname', 'fqdn' and 'domain' facts for use in the manifest, in particular for determining which 'node' statement applies to the client. Possible values are 'cert' (use the subject's CN in the client's certificate) and 'facter' (use the hostname that the client reported in its facts)

- *Default*: cert

node_terminus

Where to find information about nodes.

- *Default*: plain

noop

Whether puppet agent should be run in noop mode.

- *Default*: false

onetime

Run the configuration once, rather than as a long-running daemon. This is useful for interactively running puppetd.

- *Default*: false

passfile

Where puppet agent stores the password for its private key. Generally unused.

- *Default*: $privatedir/password

path

The shell search path. Defaults to whatever is inherited from the parent process.

- *Default*: none

pidfile

The pid file

- *Default*: $rundir/$name.pid

plugindest

Where Puppet should store plugins that it pulls down from the central server.

- *Default*: $libdir

pluginsignore

What files to ignore when pulling down plugins.

- *Default*: .svn CVS .git

pluginsource

From where to retrieve plugins. The standard Puppet file type is used for retrieval, so anything that is a valid file source can be used here.

- *Default*: puppet://$server/plugins

pluginsync

Whether plugins should be synced with the central server.

- *Default*: false

postrun_command

A command to run after every agent run. If this command returns a non-zero return code, the entire Puppet run will be considered to have failed, even though it might have performed work during the normal run.

preferred_serialization_format

The preferred means of serializing ruby instances for passing over the wire. This won't guarantee that all instances will be serialized using this method, since not all classes can be guaranteed to support this format, but it will be used for all classes that support it.

- *Default*: pson

prerun_command

A command to run before every agent run. If this command returns a non-zero return code, the entire Puppet run will fail.

privatedir

Where the client stores private certificate information.

- *Default*: $ssldir/private

privatekeydir

The private key directory.

- *Default*: $ssldir/private_keys

publickeydir

The public key directory.

- *Default*: $ssldir/public_keys

puppetdlockfile

A lock file to temporarily stop puppet agent from doing anything.

- *Default*: $statedir/puppetdlock

puppetdlog

The log file for puppet agent. This is generally not used.

- *Default*: $logdir/puppetd.log

puppetport

Which port puppet agent listens on.

- *Default*: 8139

queue_source

Which type of queue to use for asynchronous processing. If your stomp server requires authentication, you can include it in the URI as long as your stomp client library is at least 1.1.1

- *Default*: stomp://localhost:61613/

queue_type

Which type of queue to use for asynchronous processing.

- *Default*: stomp

rails_loglevel

The log level for Rails connections. The value must be a valid log level within Rails. Production environments normally use `info` and other environments normally use `debug`.

- *Default*: info

railslog

Where Rails-specific logs are sent

- *Default*: $logdir/rails.log

report

Whether to send reports after every transaction.

- *Default*: true

report_port

The port to communicate with the report_server.

- *Default*: $masterport

report_server

The server to send transaction reports to.

- *Default*: $server

reportdir

The directory in which to store reports received from the client. Each client gets a separate subdirectory.

- *Default*: $vardir/reports

reportfrom

The 'from' email address for the reports.

- *Default*: report@pelin.lovedthanlost.net

reports

The list of reports to generate. All reports are looked for in puppet/reports/name.rb, and multiple report names should be comma-separated (whitespace is okay).

- *Default*: store

reportserver

(Deprecated for 'report_server') The server to which to send transaction reports.

- *Default*: $server

reporturl

The URL used by the http reports processor to send reports

- *Default*: http://localhost:3000/reports

req_bits

The bit length of the certificates.

- *Default*: 2048

requestdir

Where host certificate requests are stored.

- *Default*: $ssldir/certificate_requests

rest_authconfig

The configuration file that defines the rights to the different rest indirections. This can be used as a fine-grained authorization system for puppet master.

- *Default*: $confdir/auth.conf

route_file

The YAML file containing indirector route configuration.

- *Default*: $confdir/routes.yaml

rrddir

The directory where RRD database files are stored. Directories for each reporting host will be created under this directory.

- *Default*: $vardir/rrd

rrdinterval

How often RRD should expect data. This should match how often the hosts report back to the server.

- *Default*: $runinterval

run_mode

The effective 'run mode' of the application: master, agent, or user.

- *Default*: master

rundir

Where Puppet PID files are kept.

- *Default*: $vardir/run

runinterval

How often puppet agent applies the client configuration; in seconds.

- *Default*: 1800

sendmail

Where to find the sendmail binary with which to send email.

- *Default*: /usr/sbin/sendmail

serial

Where the serial number for certificates is stored.

- *Default*: $cadir/serial

server

The server to which server puppet agent should connect

- *Default*: puppet

server_datadir

The directory in which serialized data is stored, usually in a subdirectory.

- *Default*: $vardir/server_data

servertype

The type of server to use. Currently supported options are webrick and mongrel. If you use mongrel, you will need a proxy in front of the process or processes, since Mongrel cannot speak SSL.

- *Default*: webrick

show_diff

Whether to print a contextual diff when files are being replaced. The diff is printed on stdout, so this option is meaningless unless you are running Puppet interactively. This feature currently requires the diff/lcs Ruby library.

- *Default*: false

signeddir

Where the CA stores signed certificates.

- *Default*: $cadir/signed

smtpserver

The server through which to send email reports.

- *Default*: none

splay

Whether to sleep for a pseudo-random (but consistent) amount of time before a run.

- *Default*: false

splaylimit

The maximum time to delay before runs. Defaults to being the same as the run interval.

- *Default*: $runinterval

ssl_client_header

The header containing an authenticated client's SSL DN. Only used with Mongrel. This header must be set by the proxy to the authenticated client's SSL DN (e.g., /CN=puppet.puppetlabs.com). See http://projects.puppetlabs.com/pr for more information.

- *Default*: HTTP_X_CLIENT_DN

ssl_client_verify_header

The header containing the status message of the client verification. Only used with Mongrel. This header must be set by the proxy to 'SUCCESS' if the client successfully authenticated, and anything else otherwise. See http://projects.puppetlabs.com/projects/puppet/wiki/Using_Mongrel for more information.

- *Default*: HTTP_X_CLIENT_VERIFY

ssldir

Where SSL certificates are kept.

- *Default*: $confdir/ssl

statedir

The directory where Puppet state is stored. Generally, this directory can be removed without causing harm (although it might result in spurious service restarts).

- *Default*: $vardir/state

statefile

Where puppet agent and puppet master store state associated with the running configuration. In the case of puppet master, this file reflects the state discovered through interacting with clients.

- *Default*: $statedir/state.yaml

storeconfigs

Whether to store each client's configuration. This requires ActiveRecord from Ruby on Rails.

- *Default*: false

strict_hostname_checking

Whether to only search for the complete hostname as it is in the certificate when searching for node information in the catalogs.

- *Default*: false

summarize

Whether to print a transaction summary.

- *Default*: false

syslogfacility

What syslog facility to use when logging to syslog. Syslog has a fixed list of valid facilities, and you must choose one of those; you cannot just make one up.

- *Default*: daemon

tagmap

The mapping between reporting tags and email addresses.

- *Default*: $confdir/tagmail.conf

tags

Tags to use to find resources. If this is set, then only resources tagged with the specified tags will be applied. Values must be comma-separated.

templatedir

Where Puppet looks for template files. Can be a list of colon-seperated directories.

- *Default*: $vardir/templates

thin_storeconfigs

Boolean; wether storeconfigs store in the database only the facts and exported resources. If true, then storeconfigs performance will be higher and still allow exported/collected resources, but other usage external to Puppet might not work

- *Default*: false

trace

Whether to print stack traces on some errors

- *Default*: false

use_cached_catalog

Whether to only use the cached catalog rather than compiling a new catalog on every run. Puppet can be run with this enabled by default and then selectively disabled when a recompile is desired.

- *Default*: false

usecacheonfailure

Whether to use the cached configuration when the remote configuration will not compile. This option is useful for testing new configurations, where you want to fix the broken configuration rather than reverting to a known-good one.

- *Default*: true

user

The user puppet master should run as.

- *Default*: puppet

vardir

Where Puppet stores dynamic and growing data. The default for this parameter is calculated specially, like confdir_.

- *Default*: /var/lib/puppet

yamldir

The directory in which YAML data is stored, usually in a subdirectory.

- *Default*: $vardir/yaml

zlib

Boolean; whether to use the zlib library

- *Default*: true

Chapter 62

Report Reference

This page is autogenerated; any changes will get overwritten *(last generated on Sat May 21 17:20:11 -0700 2011)*

Puppet clients can report back to the server after each transaction. This transaction report is sent as a YAML dump of the Puppet::Transaction::Report class and includes every log message that was generated during the transaction along with as many metrics as Puppet knows how to collect. See Reports and Reporting for more information on how to use reports.

Currently, clients default to not sending in reports; you can enable reporting by setting the report parameter to true.

To use a report, set the reports parameter on the server; multiple reports must be comma-separated. You can also specify none to disable reports entirely.

Puppet provides multiple report handlers that will process client reports:

http

Send report information via HTTP to the reporturl. Each host sends its report as a YAML dump and this sends this YAML to a client via HTTP POST. The YAML is the report parameter of the request."

log

Send all received logs to the local log destinations. Usually the log destination is syslog.

rrdgraph

Graph all available data about hosts using the RRD library. You must have the Ruby RRDtool library installed to use this report, which you can get from the RubyRRDTool RubyForge page.
This package may also be available as ruby−rrd or rrdtool−ruby in your distribution's package management system. The library and/or package will both require the binary rrdtool package from your distribution to be installed.

This report will create, manage, and graph RRD database files for each of the metrics generated during transactions, and it will create a few simple html files to display the reporting host's graphs. At this point, it will not create a common index file to display links to all hosts.

All RRD files and graphs get created in the rrddir directory. If you want to serve these publicly, you should be able to just alias that directory in a web server.

If you really know what you're doing, you can tune the rrdinterval, which defaults to the runinterval.

store

Store the yaml report on disk. Each host sends its report as a YAML dump and this just stores the file on disk, in the reportdir directory.

These files collect quickly – one every half hour – so it is a good idea to perform some maintenance on them if you use this report (it's the only default report).

tagmail

This report sends specific log messages to specific email addresses based on the tags in the log messages.

See the UsingTags tag documentation for more information on tags.

To use this report, you must create a tagmail.conf (in the location specified by tagmap). This is a simple file that maps tags to email addresses: Any log messages in the report that match the specified tags will be sent to the specified email addresses.

Tags must be comma-separated, and they can be negated so that messages only match when they do not have that tag. The tags are separated from the email addresses by a colon, and the email addresses should also be comma-separated.

Lastly, there is an all tag that will always match all log messages.

Here is an example tagmail.conf:

```
all: me@domain.com
webserver, !mailserver: httpadmins@domain.com
```

This will send all messages to me@domain.com, and all messages from webservers that are not also from mailservers to httpadmins@domain.com.

If you are using anti-spam controls, such as grey-listing, on your mail server you should whitelist the sending email (controlled by reportform configuration option) to ensure your email is not discarded as spam.

This page autogenerated on Sat May 21 17:20:12 –0700 2011

Chapter 63

Indirection Reference

This page is autogenerated; any changes will get overwritten *(last generated on Sat May 21 17:19:42 -0700 2011)*

This is the list of all indirections, their associated terminus classes, and how you select between them.

In general, the appropriate terminus class is selected by the application for you (e.g., puppet agent would always use the rest terminus for most of its indirected classes), but some classes are tunable via normal settings. These will have terminus setting documentation listed with them.

catalog

- **Terminus Setting**: catalog_terminus

active_record

compiler

Puppet's catalog compilation interface, and its back-end is Puppet's compiler

queue

rest

Find resource catalogs over HTTP via REST.

yaml

Store catalogs as flat files, serialized using YAML.

certificate

ca

Manage the CA collection of signed SSL certificates on disk.

file

Manage SSL certificates on disk.

rest

Find and save certificates over HTTP via REST.

certificate_request

ca

Manage the CA collection of certificate requests on disk.

file

Manage the collection of certificate requests on disk.

rest

Find and save certificate requests over HTTP via REST.

certificate_revocation_list

ca

Manage the CA collection of certificate requests on disk.

file

Manage the global certificate revocation list.

rest

Find and save certificate revocation lists over HTTP via REST.

certificate_status

file

rest

Sign, revoke, search for, or clean certificates & certificate requests over HTTP.

facts

- **Terminus Setting**: facts_terminus

active_record

couch

facter

Retrieve facts from Facter. This provides a somewhat abstract interface between Puppet and Facter. It's only somewhat abstract because it always returns the local host's facts, regardless of what you attempt to find.

inventory_active_record

memory

Keep track of facts in memory but nowhere else. This is used for one-time compiles, such as what the stand-alone puppet does. To use this terminus, you must load it with the data you want it to contain.

network_device

Retrieve facts from a network device.

rest

Find and save facts about nodes over HTTP via REST.

yaml

Store client facts as flat files, serialized using YAML, or return deserialized facts from disk.

file_bucket_file

file

Store files in a directory set based on their checksums.

rest

This is a REST based mechanism to send/retrieve file to/from the filebucket

file_content

file

Retrieve file contents from disk.

file_server

Retrieve file contents using Puppet's fileserver.

rest

Retrieve file contents via a REST HTTP interface.

file_metadata

file

Retrieve file metadata directly from the local filesystem.

file_server

Retrieve file metadata using Puppet's fileserver.

rest

Retrieve file metadata via a REST HTTP interface.

inventory

- **Terminus Setting**: inventory_terminus

yaml

Return node names matching the fact query

key

ca

Manage the CA's private on disk. This terminus *only* works with the CA key, because that's the only key that the CA ever interacts with.

file

Manage SSL private and public keys on disk.

node

Where to find node information. A node is composed of its name, its facts, and its environment.

- **Terminus Setting**: node_terminus

active_record

exec

Call an external program to get node information. See the External Nodes page for more information.

ldap

Search in LDAP for node configuration information. See the LDAP Nodes page for more information. This will first search for whatever the certificate name is, then (if that name contains a .) for the short name, then default.

memory

Keep track of nodes in memory but nowhere else. This is used for one-time compiles, such as what the stand-alone puppet does. To use this terminus, you must load it with the data you want it to contain; it is only useful for developers and should generally not be chosen by a normal user.

plain

Always return an empty node object. Assumes you keep track of nodes in flat file manifests. You should use it when you don't have some other, functional source you want to use, as the compiler will not work without a valid node terminus.

Note that class is responsible for merging the node's facts into the node instance before it is returned.

rest

This will eventually be a REST-based mechanism for finding nodes. It is currently non-functional.

yaml

Store node information as flat files, serialized using YAML, or deserialize stored YAML nodes.

report

processor

Puppet's report processor. Processes the report with each of the report types listed in the 'reports' setting.

rest

Get server report over HTTP via REST.

yaml

Store last report as a flat file, serialized using YAML.

resource

ral

rest

resource_type

parser

Return the data-form of a resource type.

rest

Retrieve resource types via a REST HTTP interface.

status

local

rest

This page autogenerated on Sat May 21 17:19:46 -0700 2011

Chapter 64

Network Reference

This page is autogenerated; any changes will get overwritten *(last generated on Sat May 21 17:20:05 –0700 2011)*

This is a list of all Puppet network interfaces. Each interface is implemented in the form of a client and a handler; the handler is loaded on the server, and the client knows how to call the handler's methods appropriately.

Most handlers are meant to be started on the server, usually within puppet master, and the clients are mostly started on the client, usually within puppet agent.

You can find the server-side handler for each interface at puppet/network/handler/<name>.rb and the client class at puppet/network/client/<name>.rb.

CA

Provides an interface for signing CSRs. Accepts a CSR and returns the CA certificate and the signed certificate, or returns nil if the cert is not signed.

:Prefix: puppetca :Side: Server :Methods: getcert

FileBucket

The interface to Puppet's FileBucket system. Can be used to store files in and retrieve files from a filebucket.

:Prefix: puppetbucket :Side: Server :Methods: addfile, getfile

FileServer

The interface to Puppet's fileserving abilities.

:Prefix: fileserver :Side: Server :Methods: describe, list, retrieve

Master

Puppet's configuration interface. Used for all interactions related to generating client configurations.

:Prefix: puppetmaster :Side: Server :Methods: getconfig, freshness

Report

Accepts a Puppet transaction report and processes it.

:Prefix: puppetreports :Side: Server :Methods: report

Runner

An interface for triggering client configuration runs.

:Prefix: puppetrunner :Side: Client :Methods: run

Status

A simple interface for testing Puppet connectivity.

:Prefix: status :Side: Client :Methods: status

This page autogenerated on Sat May 21 17:20:05 –0700 2011