puppet
labs

Puppet Docs

Welcome to the Puppet documentation site.

An downloadable version of this guide may be found on our downloads page.

Getting Started

New users should begin here.

- An Introduction to Puppet
- Supported Platforms
- Installing Puppet – from packages, source, or gems
- Configuring Puppet – includes server setup & testing
- Frequently Asked Questions

Components

Learn more about major working parts of the Puppet system.

- puppetmasterd, puppetd, puppet, & ralsh – components of the system
- Security Infrastructure / Securing Puppet – how PKI works

Extended Knowledge

Once you've learned the basics, go here next.

- Puppet Language Tutorial – all the language details
- Puppet Modules – modules make it easy to organize and share content
- Puppet File Serving – Serving files with Puppet
- Style Guide – Puppet community conventions
- Best Practices – use Puppet effectively
- Tips & Tricks
- Troubleshooting

Puppet Dashboard (Web GUI)

Puppet from a graphical perspective.

- Installing Dashboard
- Using Dashboard

Advanced Topics

Power user features.

- 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
- External Nodes – specify what your machines do using external data sources
- Scaling Puppet – general tips & tricks
- Scaling With Passenger – for Puppet 0.24.6 and later
- Scaling With Mongrel – for older versions of Puppet

Resource Types

Documentation on what Puppet can manage out of the box.

- Types By Category – documentation on resources managed out of the box
- Alphabetical Listing

Extending Puppet

Adapt Puppet to your specific requirements through extensible code.

- Writing Custom Facts
- Writing Custom Types & Providers
- Complete Resource Example – more information on custom types & providers
- Provider Development – more about providers
- Writing Custom Functions
- Plugins In Modules – where to put plugins, how to sync to clients
- REST API – reference of api accessible resources

Development Information

Learn how to work with Puppet core.

- Running Puppet from Source – preview the leading edge
- Development Life Cycle – learn how to contribute code

Auto-generated Docs

In addition to the formatted and annotated documentation above, auto-generated documentation is available for current and previous Puppet versions. Here are links to the latest versions:

- Metaparameters – these are usable on all types
- Types – all default types
- Configuration – all configuration file settings
- Functions – all built in functions
- Report – available reports

Other Resources

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

Help Improve This Document

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.

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:

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:

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 Tutorial

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)
```

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.

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

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/reductivelabs/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

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.

Configuration Guide

Once Puppet is installed, learn how to set it up for initial operation.

Open Firewall Ports On Server and Client

In order for the puppet server to centrally manage clients, you may need to open port 8140, both TCP and UDP, on the server and client machines.

Configuration Files

The main configuration file for Puppet is `/etc/puppet/puppet.conf`. A package based installation file will have created this file automatically. Unlisted settings have reasonable defaults. To see all the possible values, you may run:

```
$ puppet --genconfig
```

Configure DNS

The puppet client looks for a server named `puppet`. If you have local DNS zone files, you may want to add a CNAME record pointing to the server machine in the appropriate zone file.

```
puppet   IN   CNAME  crabcake.picnic.edu.
```

By setting up the CNAME you will avoid having to specify the server name in the configuration of each client.

See the book "DNS and Bind" by Cricket Liu et al if you need help with CNAME records. After adding the CNAME record, restart your name server. You can also add a host entry in the `/etc/hosts` file on both the server and client machines.

For the server:

```
127.0.0.1 localhost.localdomain localhost puppet
```

For the clients:

```
192.168.1.67 crabcake.picnic.edu crabcake puppet
```

If you can ping the server by the name `puppet` but Syslog (for example `/var/log/messages`) on the clients still has entries stating the puppet client cannot connect to the server, verify port 8140 is open on the server.

Puppet Language Setup

Create Your Site Manifest

Puppet is a declarative system, so it does not make much sense to speak of “executing” Puppet programs or scripts. Instead, we choose to use the word *manifest* to describe our Puppet code, and we speak of *applying* those manifests to the managed systems. Thus, a *manifest* is a text document written in the Puppet language and meant to describe and result in a desired configuration.

Puppet assumes that you will have one central manifest capable of configuring an entire site, which we call the *site manifest*. You could have multiple, separate site manifests if you wanted, though if doing this each of them would need their own puppet servers. Individual system differences can be seperated out, node by node, in the site manifest.

Puppet will start with `/etc/puppet/manifests/site.pp` as the primary manifest, so create `/etc/puppet/manifests` and add your manifest, along with any files it includes, to that directory. It is highly recommended that you use some form of version control (git, svn, etc) to keep track of changes to manifests.

Example Manifest

The site manifest can do as little or as much as you want. A good starting point is a manifest that makes sure that your sudoers file has the appropriate permissions:

```
# site.pp
file { "/etc/sudoers":
    owner => root, group => root, mode => 440
}
```

For more information on how to create the site manifest, see the tutorials listed in the Getting Started section.

Start the Central Daemon

Most sites should only need one central puppet server. Puppet Labs will be publishing a document describing best practices for scale-out and failover, though there are various ways to address handling in larger infrastructures. For now, we’ll explain how to work with the one server, and others can be added as needed.

First, decide which machine will be the central server; this is where puppetmasterd will be run.

The best way to start any daemon is using the local server’s service management system, often in the form of init scripts.

If you’re running on Red Hat, CentOS, Fedora, Debian, Ubuntu, or Solaris, the OS package already contains a suitable init script. If you don’t have one, you can either create your own using an existing init script as an example, or simply run without one (though this is not advisable for production environments).

It is also neccessary to create the puppet user and group that the daemon will use. Either create these manually, or start the daemon with the `--mkusers` flag to create them.

```
# /usr/sbin/puppetmasterd --mkusers
```

Starting the puppet daemon will automatically create all necessary certificates, directories, and files.

To enable the daemon to also function as a file server, so that clients can copy files from it, create a fileserver configuration file and restart pupetmasterd.

Verifying Installation

To verify that your daemon is working as expected, pick a single client to use as a testbed. Once Puppet is installed on that machine, run a single client against the central server to verify that everything is working appropriately. You should start the first client in verbose mode, with the `--waitforcert` flag enabled:

```
# puppetd --server myserver.domain.com --waitforcert 60 --test
```

Adding the `--test` flag causes puppetd to stay in the foreground, print extra output, only run once and then exit, and to just exit if the remote configuration fails to compile (by default, puppetd will use a cached configuration if there is a problem with the remote manifests).

In running the client, you should see the message:

```
info: Requesting certificate
warning: peer certificate won't be verified in this SSL session
notice: Did not receive certificate
```

This message will repeat every 60 seconds with the above command.

This is normal, since your server is not auto-signing certificates as a security precaution.

On your server, list the waiting certificates:

```
# puppetca --list
```

You should see the name of the test client. Now go ahead and sign the certificate:

```
# puppetca --sign mytestclient.domain.com
```

Within 60 seconds, your test client should receive its certificate from the server, receive its configuration, apply it locally, and exit normally.

By default, puppetd runs with a waitforcert of five minutes; set the value to 0 to disable this wait-polling period entirely.

Scaling your Installation

For more about how to tune Puppet for large environments, see Scaling Puppet.

Frequently Asked Questions

This document covers frequently asked questions not well covered elsewhere in the documentation or on the main website.

You may also wish to see Troubleshooting or Techniques for additional assorted information about specific technical items.

General

What license is Puppet released under?

Puppet is open source and is released under the GNU Public License, version 2 or greater.

Why does Puppet have its own language?

People often ask why Puppet does not use something like XML or YAML as the configuration format; otherwise people ask why I didn't just choose to just use Ruby as the input language.

The input format for Puppet is not XML or YAML because these are data formats developed to be easy for computers to handle. They do not do conditionals (although, yes, they support data structures that could be considered conditionals), but mostly, they're just horrible human interfaces. While some people are comfortable reading and writing them, there's a reason why we use web browsers instead of just reading the HTML directly. Also, using XML or YAML would limit the ability to make sure the interface is declarative – one process might treat an XML configuration differently from another.

As to just using Ruby as the input format, that unnecessarily ties Puppet to Ruby, which is undesirable, and Ruby provides a bit too much functionality. We believe systems administrators should be able to model their datacenters in a higher level system, and for those that need the power of Ruby, writing custom functions, types, and providers is still possible.

Can Puppet manage workstations?

Yes, Puppet can manage any type of machine. Puppet is used to manage many organizations that have a mix of laptops and desktops.

Does Puppet run on Windows?

The short answer is ‘not yet’. Windows support is slated to be available in 2010.

What size organizations should use Puppet?

There is no minimum or maximum organization size that can benefit from Puppet, but there are sizes that are more likely to benefit. Organizations with only a handful of servers are unlikely to consider maintaining those servers to be a real problem, while those that have more need to consider carefully how they eliminate manual management tasks.

My servers are all unique; can Puppet still help?

Yes.

All servers are at least somewhat unique – with different host names and different IP addresses – but very few servers are entirely unique, since nearly every one runs a relatively standard operating system. Servers are also often very similar to other servers within a single organization – all Solaris servers might have similar security settings, or all web servers might have roughly equivalent configurations – even if they're very different from servers in other organizations. Finally, servers are often needlessly unique, in that they have been built and managed manually with no attempt at retaining appropriate consistency.

Puppet can help both on the side of consistency and uniqueness. Puppet can be used to express the consistency that should exist, even if that consistency spans arbitrary sets of servers based on any type of data like operating system, data centre, or physical location. Puppet can also be used to handle uniqueness, either by allowing special provision of what makes a given host unique or through specifying exceptions to otherwise standard classes.

When is the Next Release?

There are regular feature and release updates on the Mailing List, and you can always find the latest release on the downloads page.

I have found a security issue in Puppet. Who do I tell?

Puppet Labs and the Puppet project take security very seriously. We handle all security problems brought to our attention and ensure that they are corrected within a reasonable time frame.

If you have identified an issue then please send an email to the Security mailbox \<security@puppetlabs.net> with the details.

Recent Notifications

Experience has shown that "security through obscurity" does not work. Public disclosure allows for more rapid and better solutions to security problems. In that vein, this page addresses Puppet's status with respect to various known security holes, which could potentially affect Puppet.

CVE Status CVE-2009-3564 Resolved in 0.25.2 CVE-2010-0156 Resolved in 0.25.2

Installation

What's The Minimum Version of Ruby?

Puppet is supported on all versions of Ruby from 1.8.2 on up. It will sometimes work on 1.8.1, but it's not supported on that release.

Upgrading

Should I upgrade the client or server first?

When upgrading to a new version of Puppet, always upgrade the server first. Old clients can point at a new server but you may have problems when pointing a new client at an old server.

How should I upgrade Puppet & Facter?

The best way to install and to upgrade Puppet and Facter is via your operating system's package management system. This is easier than installing them from source. If you do install them from source make sure you remove old versions including all application and library files (excepting configuration in /etc/puppet obviously) entirely before upgrading.

How do I know what's changed when I upgrade?

The best way to find out what's changed in Puppet is to read the release notes which are posted to the puppet-announce mailing list. They will tell you about new features, functions, deprecations and other changes to Puppet.

Configuration

What characters are permitted in a class name?

Alphanumeric and hyphens '-' only. Qualified variables also can't use the hyphen.

How Do I Test/Run Manifests?

Once you have Puppet installed according the the Installation Guide , just can run the puppet executable against your example:

```
puppet -v example.pp
```

How do I manage passwords on Red Hat Enterprise Linux & Fedora Core?

As described in the Type reference you need the Shadow Password Library, this is provided by the ruby-shadow package. The ruby-shadow library is available natively for fc6 (and higher) and should build on the RHEL and CentOS variants.

How do I use Puppet's graphing support?

Puppet has graphing support capable of creating graph files of the relationships between your Puppet client configurations.

The graphs are created by and on the client, so you must enable `graph=true` in your Puppet.conf and set `graphdir` to the directory where graphs should be output. The resultant files will be created in `dot` format which is readable by OmniGraffle (OS X) or graphviz. To generate a visual from the dot file in graphviz, run the following:

```
dot -Tpng /var/puppet/state/graphs/resources.dot -o /tmp/configuration.png
```

How do all of these variables, like $operatingsystem, get set?

The variables are all set by Facter. You can get a full listing of the available variables and their values by running facter by itself in a shell.:

```
# facter
```

Are there variables available other than those provided by Facter?

Puppet also provides a few in-built variables you can use in your manifests. The first is provided by the Puppet client and returns the current environment, appropriately it is called $environment.

Also available are the $server, $serverip and $serverversion variables. These contain the fully-qualified domain, IP address, and Puppet version of the server respectively.

Can I access environmental variables with Facter?

Not directly no but Facter has a special types of facts that can be set from environment variables. Any environment variable with a prefix of FACTER will be taken by Facter and converted into a fact, for example:

```
$ FACTER_FOO="bar"
$ export FACTER_FOO
$ facter | grep 'foo'
  foo => bar
```

The value of the FACTER_FOO environmental variable would now be available in your Puppet manifests as $foo with a value of 'bar'.

Why shouldn't I use autosign for all my clients?

It is very tempting to enable autosign for all nodes, as it cuts down on the manual steps required to bootstrap a new node (or indeed to move it to a new puppetmaster).

Typically this would be done with a *.mydomain.com or even * in the autosign.conf file.

This however can be very dangerous as it can enable a node to masquerade as another node, and get the configuration intended for that node. The reason for this is that the node chooses the certificate common name ('CN' - usually its fqdn, but this is fully configurable), and the puppetmaster then uses this CN to look up the node definition to serve. The certificate itself is stored, so two nodes could not connect with the same CN (eg alice.mydomain.com), but this is not the problem.

The problem lies in the fact that the puppetmaster does not make a 1-1 mapping between a node and the first certificate it saw for it, and hence multiple certificates can map to the same node, for example:

- alice.mydomain.com connects, gets node alice { } definition.
- bob.mydomain.com connects with CN alice.bob.mydomain.com, and also matches node alice { } definition.

Without autosigning, it would be apparent that bob was trying to get alice's configuration - as the puppetca process lists the full fqdn/CN presented. With autosign turned on, bob silently retrieves alices config.

Depending on your environment, this may not present a significant risk. It essentially boils down to the question 'Do I trust everything that can connect to my Puppetmaster?'.

If you do still choose to have a permanent, or semi-permanent, permissive autosign.conf, please consider doing the following:

- Firewall your puppetmaster - restrict port tcp/8140 to only networks that you trust.
- Create puppetmasters for each 'trust zone', and only include the trusted nodes in that Puppet masters manifest.
- Never use a full wildcard such as *

Tools

Here's an overview of the major tools in the Puppet solution.

Single binary

From verison 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 â– > puppet master puppetd â– > puppet agent puppet â– > puppet apply puppetca â– > puppet cert ralsh â– > puppet resource puppetrun â– > puppet kick puppetqd â– > puppet queue filebucket â– > puppet filebucket puppetdoc â– > puppet doc pi â– > 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 â– existing configuration file will work unchanged.

Manpage documentation

Additional information about the options supported by the various tools listed below are listed in the manpages for those tools. Please consult the manpages to learn more.

puppetmasterd or puppet master

Puppetmasterd is a central management daemon. In most installations, you'll have one puppetmasterd server and each managed machine will run 'puppetd'. By default, puppetmasterd runs a certificate authority, which you can read more about in the security section.

Puppetmasterd will automatically serve up puppet orders to managed systems, as well as files and templates.

The main configuration file for both puppetmasterd and puppetd/puppet is /etc/puppet/puppet.conf, which has sections for each application.

puppetd or puppet agent

Puppetd 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 then recieve a 'compiled catalog' containing the desired system configuration that should be applied as ordered by the central server. Servers only see the information that the central server tells them they should see, for instance, the server does not see configuration orders for other servers. It is then the responsibility of the puppet daemon to make the system match the orders given. Modules and custom plugins stored on the puppetmasterd server are transferred down to managed nodes automatically.

puppet or puppet apply

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

puppetca or puppet cert

The puppetca or 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.

puppetdoc or puppet doc

Puppet doc generates documentation about Puppet and also about your Puppet manifests. It allows you to document your manifests and output details of your configuration in HTML, Markdown and RDoc.

ralsh or puppet resource

Ralsh (also available in Puppet 2.6.0 onwards as `puppet resource`) is the “Resource Abstraction Layer SHell”. It can be used to try out Puppet concepts, or simply to manipulate your local system.

There are two main usage modes. 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
...
```

Ralsh, therefore, can be a handy tool for admins who have to maintain various platforms. It’s possible to use ralsh the exact same way on OS X as it is Linux; eliminating the need to remember differences between commands. You’ll also see that it aggregrates information from multiple files and shows them together in a unified context. Thus, for new Puppet users, ralsh can make a great introduction to the power of the resource abstraction layer.

facter

Clients use a library/tool called facter to provide information about the OS (version information, IP information, etc) to the central server. These variables can then be used in conditionals, string expressions, and in templates. To see a list of facts any node offers, simply run ‘facter’ on that node. Facter is a required/included part of all Puppet installations.

Security Infrastructure

Learn about Puppet’s built in security systems.

SSL

In centrally managed server context, Puppet uses SSL for security, which is the same thing your browser uses. We find this system is much easier to distribute than relying on SSH keys for bi-directional communication, and is better suited to writing applications (as opposed to just doing remote scripting).

puppetca

When a node starts up that does not have a certificate, it checks in with the configured puppetmaster (central management daemon) and sends a certificate request. Puppet can either be configured to autosign the certificate (generally not recommended for production environmnets), or a list of certificates that need to be signed can be shown with:

```
puppetca --list
```

To sign a certificate, use the following:

```
puppetca --sign servername.example.org
```

Puppetca can also be used to revoke certificates, see the puppetca manpage for further information.

Firewall details

If a firewall is present, you should open port 8140, both tcp and udp, on the server and client machines.

File Serving Configuration

Puppet’s fileserver is used to serve up files (and directories full of files) to the client. It can be configured in /etc/puppet/fileserver.conf (default path location):

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

These three options represent the only options currently available in the configuration file. The module name, somewhat obviously, goes in the brackets. The path is the only required option. The default security configuration is to deny all access, so if no allow lines are specified, the module will be configured but available to no one.

The path can contain any or all of %h, %H, and %d, which are dynamically replaced by the clientâ– s hostname, its fully qualified domain name and itâ– s domain name, respectively. All are taken from the clientâ– s SSL certificate (so be careful if youâ– 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 youare not specifying directory locations that have trailing slashes.

auth.conf

rest_authconfig = $confdir/auth.conf

The auth.conf doesn't exist by default, but puppet has some default settings that will be put in place if you don't create an auth.conf. You'll see these settings if you run your puppetmaster in debug mode and then connect with a client.

There's also an example auth.conf file in the puppet source in conf/auth.conf

http://github.com/reductivelabs/puppet/blob/master/conf/auth.conf

The ACL's (Access Control Lists) in the auth.conf are checked in order of appearance.

Supported syntax: auth.conf supports two different syntax depending on how you want to express the ACL.

Path syntax

path /path/to/resource environment envlist method methodlist authenthicated {yes|no|on|off|any} allow host|ip|*] deny host|ip*

The path is matched as a prefix. That is /file matches both /file_metadata and /file_content.

Ex:

```
path /certificate_revocation_list/ca
method find
allow *
```

will allow all nodes to access the certificates services

Regex syntax:

This is differentiated from the path syntaxby a '~'

path ~ regex environment envlist method methodlist authenthicated {yes|no|on|off|any} allow host|ip|*] deny host|ip*

The regex syntax is the same as ruby ones.

Ex:

```
path ~ .pp$
```

will match every resource ending in .pp (manifests files for instance)

```
path ~ ^/catalog/([^/]+)$
method find
allow $1
```

will allow nodes to retrieve their only their own catalog

environment:: restrict an ACL to a specific set of environments method:: restrict an ACL to a specific set of methods (find, search, save) auth:: restrict an ACL to an authenticated or unauthenticated request the default when unspecified is to restrict the ACL to authenticated requests (ie exactly as if auth yes was present).

If you want to test the REST API without worrying about ACL permissions, here's a completely permissive auth.conf file

```
path /
auth no
allow *
```

namespaceauth.conf

authconfig = $confdir/namespaceauth.conf

This file controls the http connections to the puppet agent. It is necessary to start the puppet agent with the listen true option.

There's an example namespaceauth.conf file in the puppet source in conf/auth.conf

http://github.com/reductivelabs/puppet/blob/master/conf/namespaceauth.conf

Serverless operation

In maximum security environments, or disconnected setups, it is possible to run puppet without the central management daemon. In this case, run 'puppet' locally instead of puppetd, and make sure the manifests are transferrred to the managed machine before running puppet, as there is no 'puppetmasterd' to serve up the manifests and source material.

Language Tutorial

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 Type Guides and References. The guides section will provide in depth examples and commentary, where as the references version is auto generated (and also available for past versions of Puppet). We would recommend starting with the ‘guides’.

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  => 644,
}
```

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 can be used to refer to the resource in other parts of the Puppet configuration.

For simple resources that don’t vary much, a single name is sufficient. However, what happens if a filename is different between operating systems? For these cases, Puppet allows you to specify a local name in addition to the title:

```
file { 'sshdconfig':
    name => $operatingsystem ? {
        solaris => '/usr/local/etc/ssh/sshd_config',
        default => '/etc/ssh/sshd_config',
    },
    owner => root,
    group => root,
    mode  => 644,
}
```

By using the title, which is always the same, it’s easy to refer to the file resource elsewhere in our configuration without having to repeat that OS specific logic.

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 tutorial.

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 definitions. 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.

Definitions, 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  => 644;
        '/etc/shadow':
            owner => 'root',
            group => 'root',
            mode  => 440;
    }
}
```

You’ll notice we introduced some shorthand here. This is the same as saying:

```
class unix {
    file { '/etc/passwd':
         owner => 'root',
         group => 'root',
         mode  => 644;
    }
    file { '/etc/shadow':
         owner => 'root',
         group => 'root',
         mode  => 440;
    }
}
```

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.

Qualification Of Nested Classes

Puppet allows you to nest classes inside one another as a way of achieving modularity and scoping. For example:

```
class myclass {
class nested {
    file { '/etc/passwd':
    owner => 'root',
    group => 'root',
    mode  => 644;
    }
}
}

class anotherclass {
include myclass::nested
}
```

In this example, the `nested` class inside the outer `myclass` is included as `myclass::nested` inside of the class named `anotherclass`. Order is important here. Class `myclass` must be evaluated before class `anotherclass` for this example to work properly.

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 added using the include function but rather the resulting class can then be included more like a definition:

```
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.

Definitions

Definitions 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, defines can be reused multiple times on the same system and take parameters. Suppose we want to create a define that creates source control repositories. We probably would want to create multiple repositories on the same system, so we would use a define, 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 variables can be used within the definitions. We use dollar sign ($) variables. Note the use of the variable `$title` above. This is a bit of technical knowledge, but as of Puppet 0.22.3 and later, definitions can have both a name and a title represented by the `$title` and `$name` variables respectively. By default, `$title` and `$name` are set to the same value, but you can set a `title` attribute and pass a different name as a parameter. ‘$title’ and ‘$name’ only work in defines, not in classes or other resources.

Earlier we mentioned that “metaparameters” are attributes that are available on all resource types. Defined types can also use metaparameters, for instance we can use ‘require’ inside of a definition. We can reference the values of those metaparameters using built-in variables. Here’s an example:

```
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’d know that subversion was always required for svn checkouts. However you can see from the above how it’s possible to use defined types with requires and other metaparameters.

Classes vs. Definitions

Classes and definitions are created similarly (although classes do not accept parameters), but they are used very differently.

Definitions 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 defined in the class, because there will normally be one copy of each on a given host. (This idiom is sometimes referred to as “service-package-file”).

Definitions 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, definitions 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]
```

“Manage the ntp configuration file before the ntpd service and notify 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]
```

“First, manage the ntp package, second manage the ntp configuration file, 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]
```

“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": }
```

“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 |>
```

“Manage 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.

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.

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.

Arrays

As mentioned in the class and resource examples above, Puppet allows usage of arrays in various areas. Arrays are defined in puppet look like this:

```
[ 'one', 'two', 'three' ]
```

Several type members, such as 'alias' in the 'host' definition definition accept arrays as their value. A host resource with multiple aliases would look like this:

```
host { 'one.example.com':
    alias  => [ 'satu', 'dua', 'tiga' ],
    ip     => '192.168.100.1',
    ensure => present,
}
```

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  => 600,
}
```

Hashes

Since Puppet version 2.6.0, hashes have been supported in the language. These hashes are defined like Ruby hashes using the form:

```
hash: { key1 => val1, ... }
```

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][subjey]]
```

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, components, and nodes introduce a new scope. 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.

The "in" syntax

From Puppet 2.6.0 you can also use the "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")
}
```

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 trinary 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. Here’s a simple example:

```
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. 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

`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`.

More complex expressions combining arithmetic expressions with the Boolean operators `and`, `or`, or `not` are also possible:

```
if ( $processor_count > 2 ) and (( $ram >= 16 * $gigabyte ) or ( $disksize > 1000 )) {
    include for_big_irons
} else {
    include for_small_box
}
```

See the Expression 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

You can use the characters A-Z, a-z, 0-9, dashes ('-'), and underscores in variables, resources and class names. In Puppet releases prior to 0.24.6, you cannot start a class name with a number.

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: +, -, /, *, <<, >>
- and in Puppet 0.25.0 and later, regular expression matches with the help of the regex match operator: =~ and !~

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`.

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 )
```

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.

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 `definition` or 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.

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 configured as a colon-separated list of directories in the puppetmasterd or masternd later) section of Puppet master's config file with the `modulepath` parameter:

   ```
   [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 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 must be normal words, matching `[-\\w+]` (letters, numbers, underscores and dashes), and not containing the namespace separators `::` or `/`. While it might be desirable to allow module hierarchies, for now modules cannot be nested.

In the filesystem modules are always referred to by the down-cased version of their name to prevent the potential ambiguity that would otherwise result.

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
      lib/
         puppet/
            parser/
               functions
            provider/
            type/
         facter/
      templates/
      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 downcased_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/auto.homes resolves to the file autofs/files/auto.homes in the module’s path.

Warning: This will only work if you do not have an explicit, for example autofs mount already declared in your fileserver.conf.

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 in a standalone puppet script by running puppet –modulepath path script.pp and as part of a site manifest on a puppetmaster, 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.erb") 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).

From version Puppet 0.23.1 onwards, everything under the modulepath is automatically imported into Puppet and is available to be used. This is called module autoloading. Puppet will attempt to auto-load classes and definitions from modules, so you don't have to explicitly import them. You can just include the module class or start using the definition. Note that the init.pp file will always be loaded first, so you can put all of your classes and definitions in there if you prefer.

With namespaces, some additional magic is also available. Let's say your autofs module has a class defined in init.pp but you want an additional class called craziness. If you define that class as autofs::craziness and store it in file craziness.pp under the manifests directory, then simply using something like include autofs::craziness will trigger puppet to search for a class called craziness in a file named craziness.pp in a module named autofs in the specified module paths. Hooray!

If you prefer to keep class autofs in a file named autofs.pp, create an init.pp file containing simply:

```
import "*"
```

And you will then be able to reference the class by using include autofs just as if it were defined in init.pp.

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

The Puppet File Server

Learn how to use Puppet's file serving capability.

Puppet comes with both a client and server for copying files around. The file serving function is provided as part of the central Puppet daemon, puppetmasterd, and the client function is used through the source attribute of file objects:

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

As the example implies, Puppet's fileserving function abstracts local filesystem topology by supporting fileservice "modules".

In release 0.25.0 and later you must specify your source statements in the format:

```
source => "puppet://server/modules/module_name/file"
```

Note the addition of the "modules" statement which differs from the earlier 0.24.x releases.

Specifying a path to serve and a name for the path, clients may request by name instead of by path. This provides the ability to conceal from the client unnecessary details like the local filesystem configuration.

File Format

The default location for the file service is /etc/puppet/fileserver.conf; this can be changed using the –fsconfig flag to puppetmasterd. The format of the file is almost exactly like that of rsync, although it does not yet support the full functionality of rsync. The configuration file resembles INI files, but it is not exactly the same:

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

These three options represent the only options currently available in the configuration file. The module name, somewhat obviously, goes in the brackets. The path is the only required option. The default security configuration is to deny all access, so if no allow lines are specified, the module will be configured but available to no one.

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 it's 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

There are two aspects to securing the Puppet file server: allowing specific access, and denying specific access. By default no access is allowed. There are three ways to specify a class of clients who are allowed or denied access: by IP address, by name, or a global allow using *.

If clients are not connecting to the Puppet file server directly, eg. using a reverse proxy and Mongrel (see Using Mongrel), then the file server will see all the connections as coming from the proxy server and not the Puppet client. In this case it is probably best to restrict access based on the hostname, as explained above. Also in this case you will need to allow access to machine(s) acting as reverse proxy, usually 127.0.0.0/8.

Priority

All deny statements are parsed before all allow statements, so if any deny statements match a host, then that host will be denied, and if no allow statements match a host, it will be denied.

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 anyone into a module:

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

Style Guide

Introduction

Puppet manifests, as one would expect, can be written in multiple ways. There are many variations on how to specify resources, multiple legal syntaxes, etc. These guidelines were developed at Stanford University, where complexity drove the need for a highly formalized and structured stylistic practice which is strictly adhered to by the dozen Unix admins who maintain over fifty (50) unique server models using over five hundred (500) individual manifests and over one thousand (1,000) unique classes to maintain over four hundred (400) servers (and those numbers are growing constantly as nearly half the infrastructure remains to be migrated to Puppet). With this much code and so many individuals contributing, it was imperative that every member of the team wrote Puppet manifests in a consistent and legible manner. Thus, the following Style Guide was created.

For consistent look-and-feel of Puppet manifests across the community, it is suggested that Puppet users adhere to these stylistic guidelines. These guidelines are aimed to make large manifests as legible as possible, so the style aims to columnate and align elements as neatly as possible.

Arrow Alignment

The arrow (=>) should be aligned at the column one space out from the longest parameter. Not obvious in the above example is that the arrow is only aligned per resource. Therefore, the following is correct:

```
exec {
    "blah":
        path => "/usr/bin",
        cwd  => "/tmp";
    "test":
        subscribe   => File["/etc/test"],
        refreshonly => true;
}
```

But the following is incorrect:

```
exec {
    "blah":
        path        => "/usr/bin",
        cwd         => "/tmp";
    "test":
        subscribe   => File["/etc/test"],
        refreshonly => true;
}
```

Attributes and Values

Quoting

Attributes for resources should not be double quoted but their values should be unless they are native types. For instance, the keyword `exists` as a value of the ensure attribute of a file resource should not be in double quotes. In contrast, the owner attribute of a file should have a value in double quotes as the owner will never be a native value.

Example:

```
file { "/tmp/somefile":
    ensure => exists,
    owner  => "root",
}
```

Commas

The last attribute-value pair ends with a comma even though there are no other attribute-value pairs.

Example:

```
file { "/tmp/somefile":
    ensure => exists,
    owner  => "root",
}
```

Resource Names

All resource names should be in double quotes. Although Puppet supports unquoted resource names if they do not contain spaces, placing all names in double quotes allows for consistent look-and-feel.

Incorrect:

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

Correct:

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

Resource Declaration

Single-Line statements

In cases where there is only one attribute-value pair, a resource can be declared in a single line:

```
file { "/tmp/somefile": ensure => exists }
```

Multi-Declaration resource statements

In Puppet, when multiple resources of the same type need to be defined, they can be defined within a single semi-colon separated statement. Observe the formatting in the example below:

```
file {
    "/tmp/app":
        ensure => directory;
    "/tmp/app/file":
        ensure => exists,
        owner  => "webapp";
    "/tmp/app/file2":
        ensure => exists,
        owner  => "webapp",
        mode   => 755;
}
```

Note that in this case, even though there is a file resource with only one attribute-value pair, the single line format is not used. Also note that the last attribute-value pair ends with a semicolon.

If all the file resources had only one attribute, the single line format could be used as below:

```
file {
    "/tmp/app":       ensure => directory;
    "/tmp/app/file":  ensure => file;
    "/tmp/app/file2": ensure => file;
}
```

As in the above example, parameter names should be aligned at the column one space out from the colon after the longest resource name (`/tmp/app/file2` in this example).

Symlinks

To make the intended behavior explicitly clear, the following is recommended as the correct way to specifiy symlinked files:

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

Case Statements

A well-formed example of a case select statement:

```
case $cloneid {
    "xorn": {
        include soe-workstation
        case $hostname {
            "ford",
            "holbrook",
            "trillian": {
                include support-workstation
            }
            "mw140ma",
            "mw140mb",
            "mw140mc",
            "mw140md": {
                include doc-scanner,
                    flat-scanner
            }
        }
    }
    "phoenix": {
        include soe-workstation,
            site-scripts:workstation
        case $hostname {
            "ford",
            "holbrook",
            "trillian": {
                include support-workstation
            }
            "mw140ma",
            "mw140mb",
            "mw140mc",
            "mw140md": {
                include doc-scanner,
                    flat-scanner
                }
            "mw430ma",
            "mw430mb",
            "mw430mc": {
                include doc-scanner
            }
        }
    }
}
```

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.

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 parameter recurse 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"
}
```

I Want To Manage A Directory and Purge Its Contents

Many people try to do something like this:

```
file { "/etc/nagios/conf.d":
  owner   => nagios,
  group   => nagios,
  purge   => true,
  recurse => true,
  force   => true,
}
```

It seems what most people expect to happen here is create the named directory, set the owner and group, then purge any files or directories that are not managed by puppet underneath that. But this is not the behaviour puppet will display. In fact the purge will silently fail.

The workaround is to define a source for the directory that doesn't exist.

```
file { "/etc/nagios/conf.d":
   owner   => nagios,
   group   => nagios,
   purge   => true,
   recurse => true,
   force   => true,
   source  => "puppet:///nagios/emtpy",
}
```

The fileserver context (here 'nagios') must exist and be created by the fileserver and an empty directory must also exist, and puppet will purge the files as expected.

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 parameter, 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?

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 Tutorial .

In versions prior to 0.24.6 Puppet does not support complex conditionals: if can only test for the existence of a variable, and case can only switch on a defined string.

However, a workaround is possible by using the generate() function to call out to Ruby or Perl to perform the comparison, eg:

```
if generate('/usr/bin/env', 'ruby', '-e', 'puts :true if eval ARGV[0]', "'$operatingsystem' =~ /r
     # do stuff
}
```

The Perl equivalent is:

```
if generate('/usr/bin/env', 'perl', '-e', 'print("true") if (eval shift)', "'$operatingsystem' =~
     # do stuff
}
```

Note that the generate() function is executed on the Puppet master at manifest compilation, so cannot be used client-side. Also note the quoting around the final option - this is to ensure that $operatingsystem variable is interpolated by the parser and passed to the Ruby or Perl interpreter as a string.

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.

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.

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 "puppetca –list"?

puppetca must be run as root. If you are not root then rerun the command with sudo:

```
sudo puppetca --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 puppetca --clean \$fqdn" on the Puppetmaster to
    clear the certificates.
2.  Remove the entire SSL directory of the client machine:

> 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.
```

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 apparently happen if Ruby is not compiled with IPv6 support; see the mail thread for more details. 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 parameter, which calls the init script with the status parameter. 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, no matter what the actual service state is. Puppet only uses the exit code, so interprets this as “the service is stopped”.

There are two workarounds, and one fix. If you must deal with the scripts broken behavior as is, you can modify your resource to either use the pattern parameter to look for a particular process name, or the status parameter to use a custom script to check for the service status.

The fix is to rewrite the 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.

I’m using the Nagios types but the Puppet run on the server running Nagios is starting to take ages

Note: This has heavily been improved with 0.25 and it’s not anymore a problem. The proposed workaround should be considered only on systems running puppet \< 0.25

If you are exporting Nagios objects, they will need to be collected on your Nagios server to be added to the nagios service configuration file.

Let’s say you are exporting some hundreds of nagios_service objects, the corresponding nagios_service.cfg file will be quite large. On every puppet run however, for each and every nagios_service defined, puppet will have to seek through that file to see if anything has changed. This is incredibly time consuming.

You’ll have better results using the “target” option for the nagios_service tyepe, e.g.

```
nagios_service { "check_ssh_$fqdn":
     ensure => present,
     check_command => "check_ssh",
     use => "generic-service",
     host_name => $fqdn,
     service_description => "SSH",
     target => "/etc/nagios/nagios_service.d/check_ssh_$fqdn"
}
```

This will dramatically improve your puppet run performance, as puppet only has to look in the directory for the file it needs and seek through a file with only one service definition in it.

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 http://www.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 doublequotes 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_namesystem 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 can occur when you use a manifest like:

```
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 and assign 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.

Puppet Dashboard

The Puppet Dashboard is a Puppet web interface that provides node management and reporting tools. Dashboard can be used as an external node classification tool.

Here you can learn how to install Dashboard.

Latest version of code/instructions

The latest version of Dashboard can be found at github.com along with the latest version of this install documentation. They are included here as an overview for printed documentation purposes, though we'd recommend you read the versions from github instead.

Dependencies

- ruby (built with iconv support) >= 1.8.1
- rake >= 0.8.3
- MySQL
- ruby mysql bindings (`gem install mysql` for ruby >= 1.8.6 or use your package manager's mysql-ruby package)

Installation

1. **Obtain the source:** `git clone git://github.com/reductivelabs/puppet-dashboard.git`
2. **Configure the database:** `rake install`
3. **Start the server:** `script/server`
4. **Import Reports (optional):** `rake reports:import`

RedHat and CentOS users: see Install puppet-dashboard on RedHat/CentOS 5 : Mike Zupan's Random Blog

This will start a local Puppet Dashboard server on port 3000. As a Rails application, Puppet Dashboard can be deployed in any server configuration that Rails supports. Instructions for deployment via Phusion Passenger coming soon.

Note: Puppet Dashboard is currently MySQL only. Other databases coming soon.

Reporting

Report Import

To import puppet run reports stored in /var/puppet/lib/reports:

```
rake reports:import
```

To specify a different report directory:

```
rake reports:import REPORT_DIR /path/to/your/reports
```

Live report aggregation

To enable report aggregation in Puppet Dashboard, the file `lib/puppet/puppet_dashboard.rb` must be available in Puppet's lib path. The easiest way to do this is to add `RAILS_ROOT/lib/puppet` to `$libdir` in your `puppet.conf`, where `RAILS_ROOT` is the directory containing this README. Then ensure that your puppetmasterd runs with the option `--reports puppet_dashboard`.

The puppet_dashboard report assumes that your Dashboard server is available at `localhost` on port 3000 (as it would be if you started it via `script/server`). For now, you will need to modify the constants in `puppet_dashboard.rb` if this is not the case.

External Node Tool

Puppet Dashboard functions as an external node tool. All nodes make a puppet-compatible YAML specification available for export. See bin/external_node for an example script that connects to Puppet Dashboard as an external node tool. See the instructions here for more information about external nodes.

Using Dashboard

Learn how to use Puppet dashboard.

About

Puppet Dashboard is fairly self explanatory, if you have set it up using the Installation Instructions just visit port 3000 to start using it. As Dashboard evolves, we'll have more information here about using it, as well as screenshots and advanced feature information.

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 (./modules.html).

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 -%>
```

And this snippet will print all the defined class in the catalog:

```
<% classes.each do |klass| -%>
The class <%= klass %> is defined
<% end -%>
```

Syntax Checking

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

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

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 client: You can use a special syntax called a collection, or you can use the simple function realize. Collections provide a simple syntax for marking virtual objects as real, such that they should be sent to the client. 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]
```

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 configuration parameter 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 parameters 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 parameters 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
```

Environments

Manage development, stage, and production differences.

Using Multiple Environments

As of 0.24.0, Puppet has support for multiple environments, along the same lines as Ruby on Rails. The idea behind these environments is to provide an easy mechanism for managing machines at different levels of SLA – some machines need to be up constantly and thus cannot tolerate disruptions and usually use older software, while other machines are more up to date and are used for testing upgrades to more important machines.

Puppet allows you to define whatever environments you want, but it is recommended that you stick to production, testing, and development for community consistency.

Puppet defaults to not using an environment, and if you do not set one on either the client or server, then it will behave as though environments do not exist at all, so you can safely ignore this feature if you do not need it.

Please note: Not using environments doesn’t mean that client doesn’t have an environment set. The client’s environment is per default set to production and will only be changed by changing the clients configuration or per command parameter. You can’t set it to a default value on the server side . For a more detailed discussion have a look at: environment default setting thread on the mailing list.

Goal of Environments

The main goal of a set-up split by environments could be that puppet can have different sources for modules and manifests for different environments on the same Puppet master.

For example, you could have a stable and a testing branch of your manifests and modules. You could then test changes to your configuration in your testing environment without impacting nodes in your production environment.

You could also use environments to deploy infrastructure to different segments of your network, for example a dmz environment and a core environment. You could also use environments to specify different physical locations,

Using Environments on the Puppet Master

The point of the environment is to choose which manifests, templates, and files are sent to the client. Thus, Puppet must be configured to provide environment-specific sources for this information.

Puppet environments are implemented rather simply: You add per-environment sections to the server’s puppet.conf configuration file, choosing different configuration sources for each environment. These per-environment sections are then used in preference to the main sections. For instance:

```
[main]
    manifest   = /usr/share/puppet/site.pp
    modulepath = /usr/share/puppet/modules
```

```
[development]
    manifest   = /usr/share/puppet/development/site.pp
    modulepath = /usr/share/puppet/development/modules
```

In this case, any clients in the development environment will use the `site.pp` file located in the directory `/usr/share/puppet/development` and Puppet would search for any modules under the `/usr/share/puppet/development/modules` directory.

Running with any other environment or without an environment would default to the `site.pp` file and directory specified in the `manifest` and `modulepath` values in the `[main]` configuration section.

Only certain parameters make sense to be configured per-environment, and all of those parameters revolve around specifying what files to use to compile a client's configuration. Those parameters are:

- **modulepath**: Where to look for modules. It's best to have a standard module directory that all environments share and then a per-environment directory where custom modules can be stored.
- **templatedir**: Where to look for templates. The modulepath should be preferred to this setting, but it allows you to have different versions of a given template in each environment.
- **manifest**: Which file to use as the main entry point for the configuration. The Puppet parser looks for other files to compile in the same directory as this manifest, so this parameter also determines where other per-environment Puppet manifests should be stored. With a separate module path, it should be easy to use the same simple manifest in all environments.

Note that using multiple environments works much better if you rely largely on modules, and you'll find it easier to migrate changes between environments by encapsulating related files into a module. It is recommended that you switch as much as possible to modules if you plan on using environments.

Additionally, the file server uses an environment-specific module path; if you do your file serving from modules, instead of separately mounted directories, your clients will be able to get environment-specific files.

Finally, the current environment is also available as the variable `$environment` within your manifests, so you can use the same manifests everywhere and behave differently internally depending on the environment.

Setting The Client's Environment

To specify which environment the Puppet client uses you can specify a value for the `environment` configuration variable in the client's `puppet.conf` file:

```
[puppetd]
    environment = development
```

This will inform the server which environment the client is in, here `development`.

You can also specify this on the command line:

```
# puppetd --environment=development
```

Alternatively, rather than specifying this statically in the configuration file, you could create a custom fact that set the client environment based upon some other client attribute or an external data source.

The preferred way of setting the environment is to use an external node configuration tool; these tools can directly specify a node's environment and are generally much better at specifying node information than Puppet is.

Puppet Search Path

When determining what configuration to apply, Puppet uses a simple search path for picking which value to use:

- Values specified on the command line
- Values specified in an environment-specific section
- Values specified in an executable-specific section
- Values specified in the main section

Plugins and Facts

Note that currently Plugins In Modules don't mix well with environments. Although you may put plugins such as types or facts into modules, they will still be referenced by the default modulepath, and not by the modulepath of the client environment.

If you want your plugins and facts to be part of your environment, one workaround is to create stub modules called plugins and facts in your environment modulepath and place your desired plugins and facts inside the files subdirectory of these stub modules.

Then when your client requests:

```
puppet://$server/facts
puppet://$server/plugins
```

for fact and plugin synchronisation, they will instead be redirected to the contents of:

```
$modulepath/facts/files/
$modulepath/plugins/files/
```

instead.

This allows your facts to differ depending upon your environment.

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.

{WARNING: Missing references to pages}

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.

External Nodes

Do you have an external database (or LDAP? or File?) that lists which of your machines should fulfill certain functions? Puppet's external nodes feature helps you tie that data into Puppet, so you have less data to enter and manage.

What's an External Node?

External nodes allow you to store your node definitions in an external data source. For example, a database or other similar repository. When the Puppet client connects the master queries the external node script and asks "Do you have a host called insertnamehere" by passing the name of the host as the first argument to the external nodes script.

This allows you to:

1. to avoid defining each node in a Puppet manifest and allowing a greater flexibility of maintenance.
2. potentially query external data sources (such as LDAP or asset management stores) that already know about your hosts meaning you only maintain said information in one place.

A subtle advantage of using a external nodes tool is that parameters assigned to nodes in a an external node tool are set a top scope not in the scope created by the node assignment in language. This leaves you free to set default parameters for a base node assignment and define whatever inheritance model you wish for parameters set in the children. In the end, Puppet accepts a list of parameters for the node and those parameters when set using an External Node tool are set at top scope.

How to use External Nodes

To use an external node classifier, in addition to or rather than having to define a node entry for each of your hosts, you need to create a script that can take a hostname as an argument and return information about that host for puppet to use.

You can use node entries in your manifests together with External nodes. You cannot however use external nodes and LDAP nodes together. You must use one of the two types.

Limitations of External Nodes

External nodes can't specify resources of any kind - they can only specify class membership, environments and attributes. Those classes can be in hierarchies however, so inheritance is available.

Configuring puppetmasterd

First, configure your puppetmasterd to use an external nodes script in your /etc/puppet/puppet.conf:

```
[main]
external_nodes = /usr/local/bin/puppet_node_classifier
node_terminus = exec
```

There are two different versions of External Node support, the format of the output required from the script changed drastically (and got a lot better) in version 0.23. In both versions, after outputting the information about the node, you should exit with code 0 to indicate success, if you want a node to not be recognized, and to be treated as though it was not included in the configuration, your script should exit with a non-zero exit code.

External node scripts for version 0.23 and later

Starting with version 0.23, the script must produce YAML output of a hash, which contains one or both of the keys classes and parameters. The classes value is an array of classes to include for the node, and the parameters value is a hash of variables to define. If your script doesn't produce any output, it may be called again with a different hostname, in my testing, the script would be called up to three times, first with hostname.example.com as an argument, then just with hostname, and finally with default. It will only be called with the shorter hostname or with default if the earlier run didn't produce any output:

```
#!/bin/sh
# Super-simple external_node script for versions 0.23 and later
cat <<"END"
---
classes:
  - common
  - puppet
  - dns
  - ntp
environment: production
parameters:
  puppet_server: puppet.example.com
  dns_server: ns.example.com
  mail_server: mail.example.com
END
exit 0
```

In addition to these options you can also access fact values in your node classifier scripts. Before the classifier is called the $vardir/yaml/facts/ directory is populated with a YAML file containing fact values. This file can be queried for fact values.

This example will produce results basically equivalent to this node entry:

```
node default {
    $puppet_server = 'puppet.example.com'
    $dns_server = 'ns.example.com'
    $mail_server = 'mail.example.com'
    include common, puppet, dns, ntp
}
```

The resulting node will also be located in the "production" environment.

In both versions, the script should exit with code 0 after producing the desired output. Exit with a non-zero exit code if you want the node to be treated as though it was not found in the configuration.

External node scripts for versions before 0.23

Before 0.23, the script had to output two lines, the first was a parent node, and the second was a list of classes:

```
#!/bin/sh
# Super-simple external_node script for versions of puppet prior to 0.23
echo "basenode"
echo "common puppet dns ntp"
exit 0
```

This sample script is essentially the same as this node definition:

```
node default inherits basenode {
  include common, puppet, dns, ntp
}
```

Scaling Puppet

Tune Puppet for maximum performance in large environments.

Are you using the default webserver?

The default web server used to enable Puppet's web services connectivity is "WEBrick", which is essentially a reference implementation, and is not very fast. Switching to a more efficient web server implementation such as Passenger or Mongrel will allow for serving many more nodes concurrently from the same server. This performance tweak will offer the most immediate benefits. 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 'puppetd' 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 Func could be used to launch 'puppetd' 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 need these features, it is possible to rsync (or otherwise transfer) puppet manifests and data to each individual node, and then run puppet locally, for instance, from 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 is not going to be as efficient as rsync or NFS, so it should not be used to serve up large directories. For small directories, however, there is no problem in using it. This will result in performance improvements on both the client and server.

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,
    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_passeng
   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.

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.

Resource Types

Resources Types are the building blocks of your Puppet configuration.

- This is an alphabetical listing of the standard types (see the categorized listing)

Introduction

Terms

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
: The 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.

Standard Types

- augeas
- computer
- cron
- exec
- file

- filebucket
- group
- host
- index
- k5login
- macauthorization
- mailalias
- maillist
- mcx
- mount
- nagios types
- notify
- package
- resources
- schedule
- selinux types
- service
- ssh types
- tidy
- user
- yumrepo
- zfs
- zone
- zpool

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):

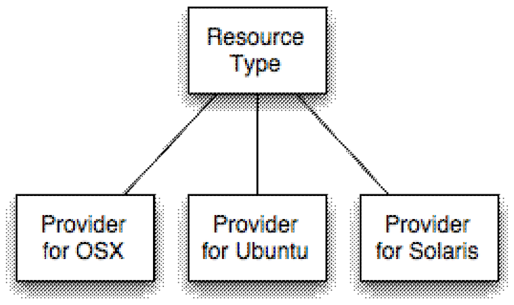
- factpath ($vardir/facts): Where Puppet should look for facts. Multiple directories should be colon-separated, like normal PATH variables. By default, this is set to the same value as factdest, but you can have multiple fact locations (e.g., you could have one or more on NFS).
- factdest ($vardir/facts): 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.

Custom Types

Learn how to create your own custom types & providers in Puppet

Organizational Principles



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 Tutorial, 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 Development_Provider_Development|Development/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.

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

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).

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 server (the host running puppetmasterd) 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.

To write functions, you’ll want to have a fundamental understanding of the Ruby programming language, since all functions must be written in that language.

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 – *statements* and *rvalues*. The difference is in
whether the function is supposed to return a value or not. You must
declare if your function is an "rvalue" by passing :type =>
rvalue when creating the function (see the examples below).

- The name of the file you put your function into must be the same as the name of function, otherwise it won't get automatically loaded. This "will" bite you some day.
- 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 loaded from files with a .rb extension in the following locations:

- $libdir/puppet/parser/functions
- $moduledir/$modulename/plugins/puppet/parser/functions
- puppet/parser/functions sub-directories in your Ruby $LOAD_PATH

For example, if default libdir is /var/puppet/lib, then you would put your functions in /var/puppet/lib/puppet/parser/functions.

The file name is derived from the name of the function that Puppet is trying to run. So, let's say that /usr/local/lib/site_ruby is in your machine's $LOAD_PATH (this is certainly true on Debian/Ubuntu systems, I'd expect it to be true on a lot of other platforms too), you can create the directory /usr/local/lib/site_ruby/puppet/parser/functions, then put the code for the my_function function in /usr/local/lib/site_ruby/puppet/parser/functions/my_function.rb, and it'll be loaded by the Puppetmaster when you first use that function.

Baby’s 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 /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!")
```

Now, before Luke has a coronary – that is not a good example of a useful function. I can’t imagine why you would want to be able to do this in Puppet in real life. This is purely an example of how a function *can* be written.

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. Hence it must be used on its own. 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, 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

“But damnit”, you say, “now you’ve got this idea of splaying my cron jobs on different machines, and I just ”have” to do this now. You can’t leave me hanging like this.” Well, it can be done. The trick is to tie your minute value to something that’s invariant in time, but different across machines. Personally, I like the MD5 hash of the hostname, modulo 60, or perhaps the IP address of the host, converted to an integer, modulo 60. Neither of them will guarantee uniqueness, but you can’t really expect that with a range of no more than 60 anyway.

But how do you get at the hostname or IP address of the client machine? “You already told us that functions are run on the server, so that’s your idea up in smoke then.” Aaah, but we have *facts*. Not opinions, but cold hard facts. And we can use them 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(‘name’).

Accessing Files

If your function will be accessing files, then you need to let the parser know that it must recompile the configuration if that file changes. In 0.23.2, this can be achieved by adding this to the function:

```
self.interp.newfile($filename)
```

In future releases, this will change to parser.watch_file($filename).

Finally, an example. This function takes a filename as argument and returns the last line of that file as its value:

```
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
```

A directory name may also be passed. The exact behaviour may be platform-dependent, but on my GNU/Linux system, this caused it to watch for files being added, removed, or modified in that directory.

Note that there may be a delay before Puppet picks up the changed file, so if your Puppet clients are contacting the master very regularly (I test with a 3 second delay), then it may be a few runs through the configuration before the file change is detected.

Calling Functions from Functions

Functions can be accessed from other functions by prefixing them with “function” and underscore.

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.

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. It 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.

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.

Usage for Client Custom Facts and Types

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.

To enable module distribution you need to make changes on both the Puppet master and the clients.

If you have existing plugins you were syncing from an earlier version of Puppet , we'll need to perform a few steps to get things converted to using the new system. On the Puppet master, if you have an existing plugins section in your fileserver.conf, get rid of the path parameter (if you leave the path parameter in place, then the mount will behave like any other fileserver mount). Move your existing plugins from the directory specified in that path into the modules for which they are relevant. If you do not have any modules defined or the types and facts are not relevant to any particular module you can create a generic module to hold all custom facts and types. It is recommended that you name this module `custom`.

While ordinarily in Puppet, you do not need to know Ruby, plugins are in fact Ruby modules. Ruby libraries in your plugins directories behave like Ruby libraries in any other (system) Ruby lib directory, and their paths need to match whatever Ruby would normally look for.

Module structure for 0.24.x (outdated)

For example, Ruby expects Puppet resource types to be in $libdir/puppet/type/$type.rb, so for modules you would put them here:

```
<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
```

Module structure for 0.25.x

In 0.25.0 and later releases, Puppet changes uses 'lib' for the name of the plugins directory.

This change was introducued in 0.25.0 and modules with an outdated `plugins` directory name will generate a deprecation warning. The plugins directory will be formally deprecated in the Rowlf release of Puppet. This deprecation changes your module paths to:

```
<MODULEPATH>/<module>/lib/puppet/type
```

For providers, place them in:

```
<MODULEPATH>/<module>/lib/puppet/provider
```

Similarly, Facter facts belong in the facter subdirectory of the library directory:

```
<MODULEPATH>/<module>/lib/facter
```

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/facter
```

Enabling Pluginsync

After setting up the directory structure, we then need to turn on pluginsync in our puppet.conf configuration file. If we're delivering Facter facts we also need to specify the factpath option, so that the facts dropped by pluginsync are loaded by Puppet:

```
[main]
pluginsync = true
factpath = $vardir/lib/facter
```

Additional options required for versions prior to 0.24.4

For versions earlier than 0.24.4 you may need to set the pluginsource and plugindest values. In 0.24.4 onwards you should remove these values as the defaults are now preferable. If you do need to set these values then try something like:

```
[main]
pluginsync=true
factsync=true
pluginsource = puppet://$server/plugins
plugindest = $vardir/lib
factpath = $vardir/lib/facter
```

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>/plugins/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/plugins/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 plugins directory and deposit it within the correct location in $libdir. The next invocation of puppetmasterd by a client will autoload the custom function.

As always custom functions are loaded once by puppetmasterd. Simply replacing a custom function with a new version will not cause puppetmasterd to automatically reload the function. You must restart puppetmasterd.

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. The easiest way to do this is to have puppet agent already authorized with a certificate that has been signed by the puppet master and use that certificate to connect, or for testing to set the security policy so that any request from anywhere is allowed.

The security policy for the API can be controlled through the rest_authconfig file, and specifically for the nodes running puppet agent through the namespaceauth file.

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
```

Most of this command is just setting the appropriate ssl certificates, which will be different depending on where your ssldir is and your node name. For simplicity, lets look at this command without the certificate related options, which I'll assume you're either passing in as above or changing the security policy so that you don't need to be authenticated. If you are changing the security policy curl will want a -k or –insecure option to connect to an https address without certificates.

```
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
```

puppet master and puppet agent shared REST API Reference

Certificate

GET `/certificate/{ca, other}`

```
curl -k -H "Accept: s" https://puppetmaster:8140/production/certificate/ca
curl -k -H "Accept: s" https://puppetcleint:8139/production/certificate/puppetclient
```

puppet master REST API Reference

Authenticated Resources (valid, signed certificate required)

Catalogs

GET `/{environment}/catalog/{node certificate name}`

```
curl -k -H "Accept: pson" https://puppetmaster:8140/production/catalog/myclient
```

Certificate Revocation List

GET `/certificate_revocation_list/ca`

```
curl -k -H "Accept: s" https://puppetmaster:8140/production/certificate/ca
```

Certificate Request

GET `/{environment}/certificate_requests/{anything}` GET `/{environment}/certificate_request/{node certificate name}`

```
curl -k -H "Accept: yaml" https://puppetmaster:8140/production/certificate_requests/all
curl -k -H "Accept: yaml" https://puppetmaster:8140/production/certificate_request/puppetclient
```

Reports - Submit a report

PUT `/{environment}/report/{node certificate name}`

```
curl -k -X PUT -H "Content-Type: text/yaml" -d "{key:value}" https://puppetclient:8139/production
```

File Server

GET `/file{metadata, content, bucket}/{file}` File serving is covered in more depth on the wiki (http://projects.puppetlabs.com/projects/puppet/wiki/File_Serving_Configuration)

Node - Returns the Facts for the specified node

GET `/{environment}/node/{node certificate name}`

```
curl -k -H "Accept: yaml" https://puppetmaster:8140/production/node/puppetclient
```

Status - Just used for testing

GET `/{environment}/status/{anything}`

```
curl -k -H "Accept: pson" https://puppetmaster:8140/production/certificate_request/puppetclient
```

puppet agent REST API Reference

puppet agent is by default 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. The namespaceauth file must also exist, and the rest_authconfig must allow access to these resources, which isn’t done by default.

Facts

GET `/{environment}/facts/{anything}`

```
curl -k -H "Accept: yaml" https://puppetclient:8139/production/facts/{anything}
```

Run - Cause the client to update like puppetrun or puppet kick

PUT `/{environment}/run/{node certificate name}`

```
curl -k -X PUT -H "Content-Type: text/pson" -d "{}" https://puppetclient:8139/production/run/{any
```

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/reductivelabs/facter
$ git clone git://github.com/reductivelabs/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
```

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.

Contribute

We welcome community contributions to the documentation.

Here's how you can help.

Providing Feedback

You don't have to write documentation to help us out; you can submit feedback from any page by using the "Feedback" tab to the left, or wherever you see this:

See an inaccuracy or think we're missing something? Let us know!

Feedback is great, and we'd like as much as possible… but if you know what needs to be added (to something you see, or based on other user feedback), here's how to contribute directly:

Making a Contribution

Version control for the project is handled with Git.

The URL of the repository is: http://github.com/reductivelabs/puppet-docs

We recommend using a GitHub account to contribute to this project – but we also accept git patches. Read below for more information.

Fork the project

If you're using Github, fork our repository, and clone your fresh repository:

```
$ git clone git@github.com:yourname/puppet-docs.git
```

If you're not using GitHub, just clone our copy directly (you can push to your own remote host or provide git patches later):

```
$ git clone git://github.com/reductivelabs/puppet-docs.git
```

Learn how add documentation

Read the README and README_WRITING in the source.

Make your changes

Add your documentation fixes.

If you modify any of the generator (Ruby) code, make sure you provide passing tests that cover your changes.

Commit and Push

- If you're using GitHub (or your own hosted repository), push to a remote branch.
- If you're not working with a remote, generate a patch of your changes for the next step.

If you need a refresher on how to commit and work with remote repositories in Git, you may want to visit GitHub's articles and screencasts.

Submit a Ticket

Visit the Puppet Documentation Project and submit a ticket. You'll need to create a Redmine account if you don't already have one.

In your ticket, provide:

1. Any additional background on your change.
2. The versions of Puppet (or supporting project) to which it pertains.
3. If using GitHub or another remote host, the URL to the branch you're submitting (so we can pull it); if not, one or more attached git patches.

We'll get back to you on your contribution as soon as possible. Thanks!

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).

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.

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 server (the host running puppetmasterd) 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.

To write functions, you'll want to have a fundamental understanding of the Ruby programming language, since all functions must be written in that language.

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 – *statements* and *rvalues*. The difference is in whether the function is supposed to return a value or not. You must declare if your function is an "rvalue" by passing :type => rvalue when creating the function (see the examples below).
- The name of the file you put your function into must be the same as the name of function, otherwise it won't get automatically loaded. This "will" bite you some day.
- 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 loaded from files with a .rb extension in the following locations:

- $libdir/puppet/parser/functions
- $moduledir/$modulename/plugins/puppet/parser/functions
- puppet/parser/functions sub-directories in your Ruby $LOAD_PATH

For example, if default libdir is /var/puppet/lib, then you would put your functions in /var/puppet/lib/puppet/parser/functions.

The file name is derived from the name of the function that Puppet is trying to run. So, let’s say that /usr/local/lib/site_ruby is in your machine’s $LOAD_PATH (this is certainly true on Debian/Ubuntu systems, I’d expect it to be true on a lot of other platforms too), you can create the directory /usr/local/lib/site_ruby/puppet/parser/functions, then put the code for the my_function function in /usr/local/lib/site_ruby/puppet/parser/functions/my_function.rb, and it’ll be loaded by the Puppetmaster when you first use that function.

Baby’s 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 /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!")
```

Now, before Luke has a coronary – that is not a good example of a useful function. I can’t imagine why you would want to be able to do this in Puppet in real life. This is purely an example of how a function *can* be written.

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. Hence it must be used on its own. 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, 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

“But damnit”, you say, “now you’ve got this idea of splaying my cron jobs on different machines, and I just ”have” to do this now. You can’t leave me hanging like this.” Well, it can be done. The trick is to tie your minute value to something that’s invariant in time, but different across machines. Personally, I like the MD5 hash of the hostname, modulo 60, or perhaps the IP address of the host, converted to an integer, modulo 60. Neither of them will guarantee uniqueness, but you can’t really expect that with a range of no more than 60 anyway.

But how do you get at the hostname or IP address of the client machine? “You already told us that functions are run on the server, so that’s your idea up in smoke then.” Aaah, but we have *facts*. Not opinions, but cold hard facts. And we can use them 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(‘name’).

Accessing Files

If your function will be accessing files, then you need to let the parser know that it must recompile the configuration if that file changes. In 0.23.2, this can be achieved by adding this to the function:

```
self.interp.newfile($filename)
```

In future releases, this will change to parser.watch_file($filename).

Finally, an example. This function takes a filename as argument and returns the last line of that file as its value:

```
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
```

A directory name may also be passed. The exact behaviour may be platform-dependent, but on my GNU/Linux system, this caused it to watch for files being added, removed, or modified in that directory.

Note that there may be a delay before Puppet picks up the changed file, so if your Puppet clients are contacting the master very regularly (I test with a 3 second delay), then it may be a few runs through the configuration before the file change is detected.

Calling Functions from Functions

Functions can be accessed from other functions by prefixing them with “function” and underscore.

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.

Tools

Here’s an overview of the major tools in the Puppet solution.

Single binary

From verison 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 â– > puppet master puppetd â– > puppet agent puppet â– > puppet apply puppetca â– > puppet cert ralsh â– > puppet resource puppetrun â– > puppet kick puppetqd â– > puppet queue filebucket â– > puppet filebucket puppetdoc â– > puppet doc pi â– > 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 â– existing configuration file will work unchanged.

Manpage documentation

Additional information about the options supported by the various tools listed below are listed in the manpages for those tools. Please consult the manpages to learn more.

puppetmasterd or puppet master

Puppetmasterd is a central management daemon. In most installations, you’ll have one puppetmasterd server and each managed machine will run ‘puppetd’. By default, puppetmasterd runs a certificate authority, which you can read more about in the security section.

Puppetmasterd will automatically serve up puppet orders to managed systems, as well as files and templates.

The main configuration file for both puppetmasterd and puppetd/puppet is /etc/puppet/puppet.conf, which has sections for each application.

puppetd or puppet agent

Puppetd 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 then recieve a ‘compiled catalog’ containing the desired system configuration that should be applied as ordered by the central server. Servers only see the information that the central server tells them they should see, for instance, the server does not see configuration orders for other servers. It is then the responsibility of the puppet daemon to make the system match the orders given. Modules and custom plugins stored on the puppetmasterd server are transferred down to managed nodes automatically.

puppet or puppet apply

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

puppetca or puppet cert

The puppetca or 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.

puppetdoc or puppet doc

Puppet doc generates documentation about Puppet and also about your Puppet manifests. It allows you to document your manifests and output details of your configuration in HTML, Markdown and RDoc.

ralsh or puppet resource

Ralsh (also available in Puppet 2.6.0 onwards as `puppet resource`) is the “Resource Abstraction Layer SHell”. It can be used to try out Puppet concepts, or simply to manipulate your local system.

There are two main usage modes. 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
...
```

Ralsh, therefore, can be a handy tool for admins who have to maintain various platforms. It's possible to use ralsh the exact same way on OS X as it is Linux; eliminating the need to remember differences between commands. You'll also see that it aggregrates information from multiple files and shows them together in a unified context. Thus, for new Puppet users, ralsh can make a great introduction to the power of the resource abstraction layer.

facter

Clients use a library/tool called facter to provide information about the OS (version information, IP information, etc) to the central server. These variables can then be used in conditionals, string expressions, and in templates. To see a list of facts any node offers, simply run 'facter' on that node. Facter is a required/included part of all Puppet installations.

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 configured as a colon-separated list of directories in the puppetmasterd or masternd later) section of Puppet master's config file with the `modulepath` parameter:

   ```
   [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 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 must be normal words, matching `[-\\w+]` (letters, numbers, underscores and dashes), and not containing the namespace separators `::` or `/`. While it might be desirable to allow module hierarchies, for now modules cannot be nested.

In the filesystem modules are always referred to by the down-cased version of their name to prevent the potential ambiguity that would otherwise result.

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
      lib/
         puppet/
            parser/
               functions
            provider/
            type/
         facter/
      templates/
      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 downcased_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/auto.homes resolves to the file autofs/files/auto.homes in the module’s path.

Warning: This will only work if you do not have an explicit, for example autofs mount already declared in your fileserver.conf.

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 in a standalone puppet script by running puppet –modulepath path script.pp and as part of a site manifest on a puppetmaster, 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.erb”) 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).

From version Puppet 0.23.1 onwards, everything under the modulepath is automatically imported into Puppet and is available to be used. This is called module autoloading. Puppet will attempt to auto-load classes and definitions from modules, so you don’t have to explicitly import them. You can just include the module class or start using the definition. Note that the init.pp file will always be loaded first, so you can put all of your classes and definitions in there if you prefer.

With namespaces, some additional magic is also available. Let’s say your autofs module has a class defined in init.pp but you want an additional class called craziness. If you define that class as autofs::craziness and store it in file craziness.pp under the manifests directory, then simply using something like include autofs::craziness will trigger puppet to search for a class called craziness in a file named craziness.pp in a module named autofs in the specified module paths. Hooray!

If you prefer to keep class autofs in a file named autofs.pp, create an init.pp file containing simply:

```
import "*"
```

And you will then be able to reference the class by using include autofs just as if it were defined in init.pp.

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

Resource Types

Resources Types are the building blocks of your Puppet configuration.

- This is a categorized listing of the standard types (see the alphabetical listing)

Introduction

Terms

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
: The 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.

Standard Types

Access & Permissions

- file
- group
- user

- k5login
- macauthorization (OSX-only)
- mcx (OSX-only)
- See the ssh types

Packaging

- package
- yumrepo

Services

- service
- cron

Filesystems and Containers

- file
- mount
- zfs
- zone (Solaris-only)
- zpool

Hosts

- host
- computer (OSX-only)

Configuration Files

- augeas

Mail

- mailalias
- maillist

Monitoring

- See the nagios types

Security Policy

- See the selinux types (SELinux-only)

Executing Scripts

- exec

augeas

Apply the changes (single or array of changes) to the filesystem via the augeas tool.

- Safely manipulate configuration files
- Supports augeas' domain-specific language to describe file formats

Requirements

- augeas
- The ruby-augeas binding

Platforms

Same as augeas.

Version Compatibility

This section has not been completed.

Examples

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",
}
```

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

`set [PATH] [VALUE]`
: Sets the value `VALUE` at loction `PATH`

rm

`rm [PATH]`
: Removes the node at location `PATH`

You can also use the synonym `remove`.

clear

`clear [PATH]`
: Keeps the node at PATH, but removes the value

ins

`ins [LABEL] [WHERE] [PATH]`
: Inserts an empty node LABEL either `WHERE={before|after} PATH`

You can also use the synonym `insert`.

context

Optional context path. This value is pre-pended to the paths of all changes if the path is relative. So a path specified as `/files/foo` will not be prepended with the context while `files/foo` will be prepended

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

load_path

Optional colon separated list of directories; these directories are searched for schema definitions

name

The name of this task. Used for uniqueness.

This is the `namevar` for this resource type.

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] == [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

Supports the following features:

`execute_changes`
: Actually make the changes

`need_to_run?`
: If the command should run

`parse_commands`
: Parse the command string

returns

The expected return code from the augeas command.

This is used internally by Puppet and 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.

NOTE: Optional, defaults to `false`. Valid values are `true` and `false`.

computer

Computer object management using DirectoryService on OS X.

- Manages Computer objects in the local directory service domain.
- This type primarily exists to create localhost Computer objects to which MCX policy can then be attached.

Requirements

No additional requirements.

Platforms

Only supported on OSX (`$operatingsystem == 'darwin'`)

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. If you wish to manage `/etc/hosts` on Mac OSX,then simply use the host type as per other platforms.

Version Compatibility

This section has not been completed.

Examples

This section has not been completed.

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.

This is the `namevar` for this resource type.

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

realname

The ‘long’ name of the computer record.

cron

Installs and manages cron jobs.

Requirements

- A `crontab` executable.

Platforms

Supported on any platform with `crontab`.

Examples

```
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

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).

`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.

This is the `namevar` for this resource type.

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

Special schedules.

Only supported on FreeBSD.

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.

Introduction

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 also 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.

It is worth noting that `exec` is special, in that it is not currently considered an error to have multiple `exec` instances with the same name. This was done purely because it had to be this way in order to get certain functionality, but it complicates things. In particular, you will not be able to use `exec` instances that share their commands with other instances as a dependency, since Puppet has no way of knowing which instance you mean.

For example:

```
# defined in the production class
exec { "make":
    cwd => "/prod/build/dir",
    path => "/usr/bin:/usr/sbin:/bin"
}

. etc. .

# defined in the test class
exec { "make":
    cwd => "/test/build/dir",
    path => "/usr/bin:/usr/sbin:/bin"
}
```

Any other type would throw an error, complaining that you had the same instance being managed in multiple places, but these are obviously different images, so `exec` had to be treated specially.

It is recommended to avoid duplicate names whenever possible.

Note 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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Parameters

`command`

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.

This is the `namevar` for this resource type.

`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.

`env`

This parameter is deprecated. Use 'environment' instead.

This section has not been completed.

`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, eg:

```
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.

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 any negative number to disable the timeout. The time is specified in seconds.

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.

- Sets ownership and permissions.
- Supports creation of both files and directories.
- Retrieves entire files from remote servers.

Background

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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Examples

Here is a simple example:

```
file { "/var/www/my/file":
    source => "/path/in/nfs/or/something",
    mode   => 666
}
```

There are also a number of additional, more complete examples below.

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 puppetmasterd 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

How to check whether a file has changed. This state is used internally for file copying, but it can also be used to monitor files somewhat like Tripwire without managing the file contents in any way. You can specify that a file's checksum should be monitored and then subscribe to the file from another object and receive events to signify checksum changes, for instance.

There are a number of checksum types available including MD5 hashing (and an md5lite variation that only hashes the first 500 characters of the file.

The default checksum parameter, if checksums are enabled, is md5. Valid values are md5, md5lite, timestamp, mtime, time. Values can match /^\{md5|md5lite|timestamp|mtime|time\}/.

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.

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 be considered to be a symlink. For instance, the following text creates a link:

```
# Useful on solaris
file { "/etc/inetd.conf":
    ensure => "/etc/inet/inetd.conf"
}
```

You can make relative links:

```
# Useful on solaris
file { "/etc/inetd.conf":
    ensure => "inet/inetd.conf"
}
```

If you need to make a relative link to a file named the same as one of the valid values, you must prefix it with `./` or something similar.

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 `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.

owner

To whom the file should belong. Argument can be user name or user ID.

path

The path to the file to manage. Must be fully qualified.

This is the `namevar` for this resource type.

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. 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`).

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/module/sendmail.cf"
    }
}
```

You can also leave out the server name, in which case `puppetd` will fill in the name of its configuration server and `puppet` 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`) and the client demon (`puppetd`) differs slightly: `puppet` will look such a file up on the module path on the local host, whereas `puppetd` 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 => [
        "/nfs/files/file.$host",
        "/nfs/files/file.$operatingsystem",
        "/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. Valid values are `notlink`. Values can match `/./`.

type

A read-only state to check the file type.

filebucket

Defines a repository for backing up files.

Background

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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Parameters

`name`

The name of the filebucket.

This is the `namevar` for this resource type.

`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.

Platforms

Supported on all platforms, though features differ. See providers below for more information.

Version Compatibility

This section has not been completed.

Examples

This section has not been completed.

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`.

`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.

`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.

This is the `namevar` for this resource type.

`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.

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.

Provider	manages_members
directoryservice	**X**
groupadd	
ldap	
pw	

Available providers are:

`directoryservice`

Group management using DirectoryService.

Required binaries: * `/usr/bin/dscl`.

Default for `operatingsystem` == `darwin`.

Can manage members of the group (supports the `manages_members` feature).

`groupadd`

Group management via `groupadd` and its ilk.

The default for most platforms

Required binaries:

- `groupmod`
- `groupdel`
- `groupadd`

`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.

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 next available GID.

`pw`

Group management via `pw`.

Only works on FreeBSD.

Required binaries:

- `/usr/sbin/pw`

NOTE: Default for `operatingsystem` == `freebsd`.

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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Examples

```
host { 'peer':
    ip => '192.168.1.121',
    host_aliases => [ "foo", "bar" ],
    ensure => 'present',
}
```

Parameters

`host_aliases`

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

`ensure`

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

`ip`

The host's IP address, IPv4 or IPv6.

`name`

The host name.

This is the `namevar` for this resource type.

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

The default provider.

target

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

k5login

Manage the `.k5login` file for a user.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Examples

```
k5login { "/tmp/k5login-example":
    principals => ["test1@example", "test2@example"]
}
```

Parameters

Specify the full path to the `.k5login` file as the name and an array of principals as the property principals.

`ensure`

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

`mode`

Manage the k5login file's mode

`path`

The path to the file to manage. Must be fully qualified.

This is the `namevar` for this resource type.

`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 resource type.

macauthorization

Manage the Mac OS X authorization database.

Background

See the Apple Security Service page for more information.

Requirements

No additional requirements.

Platforms

OSX is the only supported platform.

Version Compatibility

This section has not been completed.

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`

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 ruleâ– s key to match the rule with a right. Wildcard keys end with a â– .â– . The generic rule has an empty key value. Any rights that do not match a specific rule use the generic rule.

This is the `namevar` for this resource type.

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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Examples

This section has not been completed.

Parameters

`ensure`

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

`name`

The alias name.

This is the `namevar` for this resource type.

`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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Examples

This section has not been completed.

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.

This is the `namevar` for this resource type.

`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:

- `newlist`
- `/var/lib/mailman/mail/mailman`
- `list_lists`
- `rmlist`.

`webserver`

The name of the host providing web archives and the administrative interface.

mcx

MCX object management using DirectoryService on OS X.

Background

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 `ralsh mcx` then deploying this to other machines.

Requirements

No additional requirements.

Platforms

Only supported on OSX.

Version Compatibility

This section has not been completed.

Examples

This section has not been completed.

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>
```

`ds_type` is capitalized and plural in the `dscl` command.

`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.

This is the `namevar` for this resource type.

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.

Required binaries:

- `/usr/bin/dscl`

This provider was contributed by Jeff McCune

mount

Manages mounted filesystems

- Mounts/unmounts filesystems.
- Persists mount information in the mount table.

Background

The actual behavior depends on the value of the `ensure` parameter.

Requirements

No additional requirements.

Platforms

Supported on all platforms with the `mount` and `umount` executables needed by the provider.

Version Compatibility

This section has not been completed.

Examples

This section has not been completed.

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 platforms + 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 `present` 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. Valid values are `present` (also called `unmounted`), `absent`, `mounted`.

If a `mount` receives an event from another resource, it will try to remount the filesystems if `ensure` is set to `mounted`.

fstype

The mount type. Valid values depend on the operating system.

name

The mount path for the mount.

This is the `namevar` for this resource type.

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`.

This section has not been completed.

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`

`remounts`

Whether the mount can be remounted with:

```
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.

notify

Sends an arbitrary message to the puppetd run-time log.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Parameters

`message`

The message to be sent to the log.

`name`

An arbitrary tag for your own reference; the name of the message.

This is the `namevar` for this resource type.

`withpath`

Whether to not to show the full object path. Valid values are `true`, `false`.

Package

Manages software installation and updates

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.

Features

- **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	installable	purgeable	uninstallable	upgradeable	versionable
appdmg	**X**				
apple	**X**				
apt	**X**	**X**	**X**	**X**	**X**
aptitude	**X**	**X**	**X**	**X**	**X**
aptrpm	**X**	**X**	**X**	**X**	**X**
blastwave	**X**		**X**	**X**	
darwinport	**X**		**X**	**X**	
dpkg	**X**	**X**	**X**	**X**	
fink	**X**	**X**	**X**	**X**	**X**
freebsd	**X**		**X**		
gem	**X**		**X**	**X**	**X**
hpux	**X**		**X**		
openbsd	**X**		**X**		
pkgdmg	**X**				
portage	**X**		**X**	**X**	**X**
ports	**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**

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`, `latest`. Values can match `/./`.

instance

A read-only parameter set by the package.

name

- **namevar**

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:

- **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: `installable`, `purgeable`, `uninstallable`, `upgradeable`, `versionable`.
- **aptitude**: Package management via `aptitude`. Required binaries: `/usr/bin/apt-cache`, `/usr/bin/aptitude`. Supported features: `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`.
- **darwinport**: Package management using DarwinPorts on OS X. Required binaries: `/opt/local/bin/port`. 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: `installable`, `purgeable`, `uninstallable`, `upgradeable`.
- **fink**: Package management via `fink`. Required binaries: `/sw/bin/apt-cache`, `/sw/bin/fink`, `/sw/bin/dpkg-query`, `/sw/bin/apt-get`. Supported features: `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_info`, `/usr/sbin/pkg_add`, `/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/swremove`, `/usr/sbin/swinstall`, `/usr/sbin/swlist`. Default for `operatingsystem` == `hp-ux`. Supported features: `installable`, `uninstallable`.
- **openbsd**: OpenBSD's form of `pkg_add` support. Required binaries: `pkg_info`, `pkg_add`, `pkg_delete`. Default for `operatingsystem` == `openbsd`. Supported features: `installable`, `uninstallable`.
- **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 </trac/puppet/wiki/DmgPackages>` for more detail. Required binaries: `/usr/sbin/installer`, `/usr/bin/hdiutil`, `/usr/bin/curl`. Default for `operatingsystem` == `darwin`. Supported features: `installable`.
- **portage**: Provides packaging support for Gentoo's portage system. Required binaries: `/usr/bin/update-eix`, `/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`.
- **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/bin/pkginfo`, `/usr/sbin/pkgadd`, `/usr/sbin/pkgrm`. 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 `lsbdistrelease` == `2.134` and `operatingsystem` == `redhatoelovm`. 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`, `rpm`, `python`. Default for `operatingsystem` == `fedoracentosredhat`. Supported features: `installable`, `purgeable`, `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

Manage how other resources are handled

About

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

Parameters

name

- **namevar**

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+$/`.

schedule

Determine when Puppet runs

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

- **namevar**

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.

service

About

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 make up for any inherent shortcomings (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. Or, you can just use a platform that has very good service support.

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 a special command for restarting with the `restart` attribute.

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**
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**

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 (or you can specify a specific command using the `status` parameter).

If you do not specify anything, then the service name will be looked for in the process table. Valid values are `true`, `false`.

manifest

Specify a command to config a service, or a path to a manifest to do so.

name

- **namevar**

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`.
- **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/svstat`, `/usr/bin/svc`.
    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**: FreeBSD's (and probably NetBSD?) form of `init`-style service management.

  Uses `rc.conf.d` for service enabling and disabling.

  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`

- **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 launchd jobs, the default service framework for Mac OS X, that has also been open sourced by Apple for possible 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 plists “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: `/usr/bin/sw_vers`, `/bin/launchctl`. 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/chkconfig`, `/sbin/service`. 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`.

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. 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.

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 => false,
    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.

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*, and *m* can be appended to specify *bytes*, *kilobytes*, and *megabytes*, 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.

Features

- **allows_duplicates**: The provider supports duplicate users with the same UID.
- **manages_homedir**: The provider can create and remove home directories.
- **manages_passwords**: The provider can modify user passwords, by accepting a password hash.
- **manages_solaris_rbac**: The provider can manage roles and normal users

Provider	allows_duplicates	manages_homedir	manages_passwords	manages_solaris_rbac
directoryservice			**X**	
hpuxuseradd	**X**	**X**		
ldap			**X**	
pw	**X**	**X**		
user_role_add	**X**	**X**	**X**	**X**
useradd	**X**	**X**		

Parameters

allowdupe

Whether to allow duplicate UIDs. Valid values are `true`, `false`.

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`.

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.

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

- **namevar**

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.

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:

- **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/usermod.sam`, `/usr/sam/lbin/userdel.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: `usermod`, `roledel`, `rolemod`, `userdel`, `useradd`, `roleadd`. Default for `operatingsystem` == `solaris`. Supported features: `allows_duplicates`, `manages_homedir`, `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: `usermod`, `userdel`, `useradd`. Supported features: `allows_duplicates`, `manages_homedir`.

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.

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.

yumrepo

The client-side description of a yum repository. Repository configurations are found by parsing /etc/yum.conf and the files indicated by reposdir 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 `/.*/`.

descr

A human readable description of the repository. 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 `/.*/`.

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

- **namevar**

The name of the repository.

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.

Parameters

compression

The compression property.

copies

The copies property.

ensure

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

mountpoint

The mountpoint property.

name

- **namevar**

The full name for this filesystem. (including the zpool)

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.

reservation

The reservation property.

sharenfs

The sharenfs property.

snapdir

The sharenfs property.

zone

Solaris zones.

Parameters

autoboot

Whether the zone should automatically boot. Valid values are `true`, `false`.

create_args

Arguments to the zonecfg create command. This can be used to create branded zones.

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.

name

- **namevar**

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=&lt;%= password %>
timeserver=localhost
name_service=DNS {domain_name=&lt;%= domain %>
        name_server=&lt;%= nameserver %>}
network_interface=primary {hostname=&lt;%= realhostname %>
        ip_address=&lt;%= ip %>
        netmask=&lt;%= netmask %>
        protocol_ipv6=no
        default_route=&lt;%= 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.

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_hostescalation

The Nagios type hostescalation.

Background

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/<type>.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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Parameters

`_naginator_name`

The name parameter for Nagios type hostescalation

This is the `namevar` for this resource type.

`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_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_hostgroup

The Nagios type hostgroup.

Background

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/<type>.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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

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

The name parameter for Nagios type hostgroup

This is the `namevar` for this resource type.

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_hostextinfo

The Nagios type hostextinfo.

Background

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/<type>.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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Parameters

ensure

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

host_name

The name parameter for Nagios type hostextinfo

This is the `namevar` for this resource type.

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_hostdependency

The Nagios type hostdependency.

Background

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/<type>.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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Parameters

`_naginator_name`

The name parameter for Nagios type hostdependency

This is the `namevar` for this resource type.

`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_service

The Nagios type service.

Background

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/<type>.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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Parameters

`_naginator_name`

The name parameter for Nagios type service

This is the `namevar` for this resource type.

`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_contactgroup

The Nagios type contactgroup.

Background

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/<type>.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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Parameters

`alias`

Nagios configuration file parameter.

`contactgroup_members`

Nagios configuration file parameter.

`contactgroup_name`

The name parameter for Nagios type contactgroup

This is the `namevar` for this resource type.

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_contact

The Nagios type contact.

Background

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/<type>.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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

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`

The name parameter for Nagios type contact

This is the `namevar` for this resource type.

`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_command

The Nagios type command.

Background

Background

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/<type>.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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

Parameters

`command_line`

Nagios configuration file parameter.

`command_name`

The name parameter for Nagios type command

This is the `namevar` for this resource type.

`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

Uses Naginator

target

The 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.

* *

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_host

The Nagios type host.

Background

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/<type>.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.

Requirements

No additional requirmements.

Platforms

All platforms.

Version Compatibility

This section has not been completed.

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`

The name parameter for Nagios type host

This is the `namevar` for this resource type.

`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:

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_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.

target

target

use

Nagios configuration file parameter.

* *

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.

Parameters

ensure

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

name

- **namevar**

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`.

selboolean

Manages SELinux booleans on systems with SELinux support.

- The supported booleans are any of the ones found in `/selinux/booleans/`

Parameters

name

- **namevar**

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`.

sshkey

Installs and manages ssh host keys.

- This type only knows how to install keys into `/etc/ssh/ssh_known_hosts`, and it cannot manage user authorized keys.

Parameters

alias

Any alias the host might have. Multiple values must be specified as an array. Note that this parameter has the same name as one of the metaparams; using this parameter to set aliases will make those aliases available in your Puppet scripts.

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

- **namevar**

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`).

---

ssh_authorized_key

Manages SSH authorized keys.

- Currently only type 2 keys are supported.

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

- **namevar**

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 (ie 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.