Puppet 2.7 Reference Manual

(Generated on July 01, 2013, from git revision 46784ac1656bd7b57fcfb51d0865ec7ff65533d9)

Language: Visual Index

This page can help you find syntax elements when you can't remember their names.

```
file {'ntp.conf':
  path    => '/etc/ntp.conf',
  ensure  => file,
  content => template('ntp/ntp.conf'),
  owner   => root,
  mode    => 0644,
}
```

↑ A resource declaration.

- `file`: The resource type
- `ntp.conf`: The title
- `path`: An attribute
- `'/etc/ntp.conf'`: A value; in this case, a string
- `template('ntp/ntp.conf')`: A function call that returns a value; in this case, the `template` function, with the name of a template in a module as its argument

```
package {'ntp':
  ensure => installed,
  before => File['ntp.conf'],
}
service {'ntpd':
  ensure    => running,
  subscribe => File['ntp.conf'],
}
```

↑ Two resources using the `before` and `subscribe` relationship metaparameters (which accept resource references).

```
Package['ntp'] -> File['ntp.conf'] ~> Service['ntpd']
```

↑ Chaining arrows forming relationships between three resources, using resource references.

```
$package_list = ['ntp', 'apache2', 'vim-nox', 'wget']
```

↑ A variable being assigned an array value.

```
$myhash = { key => { subkey => 'b' }}
```

↑ A variable being assigned a hash value.

```
...
content => "Managed by puppet master version ${serverversion}"
```

↑ A master-provided built-in variable being interpolated into a double-quoted string (with optional curly braces).

```
class ntp {
  package {'ntp':
    ...
  }
  ...
}
```

↑ A class definition, which makes a class avaliable for later use.

```
include ntp
require ntp
class {'ntp':}
```

↑ Declaring a class in three different ways: with the `include` function, with the `require` function, and with the resource-like syntax. Declaring a class causes the resources in it to be managed.

```
define apache::vhost ($port, $docroot, $servername = $title, $vhost_name =
'*') {
  include apache
  include apache::params
  $vhost_dir = $apache::params::vhost_dir
  file { "${vhost_dir}/${servername}.conf":
      content => template('apache/vhost-default.conf.erb'),
      owner   => 'www',
      group   => 'www',
      mode    => '644',
      require => Package['httpd'],
      notify  => Service['httpd'],
  }
}
```

↑ A defined type, which makes a new resource type available. Note that the name of the type has two namespace segments.

```
apache::vhost {'homepages':
  port    => 8081,
```

```
    docroot => '/var/www-testhost',
  }
```

↑ Declaring a defined resource (or “instance”) of the type defined above.

```
  Apache::Vhost['homepages']
```

↑ A resource reference to the defined resource declared above. Note that every namespace segment must be capitalized.

```
  node 'www1.example.com' {
    include common
    include apache
    include squid
  }
```

↑ A node definition.

```
  node /^www\d+$/ {
    include common
  }
```

↑ A regular expression node definition.

```
  import nodes/*.pp
```

↑ An import statement. Should be avoided in all but a few circumstances.

```
  # comment
  /* comment */
```

↑ Two comments.

```
  if $is_virtual == 'true' {
    warn( 'Tried to include class ntp on virtual machine; this node may be
misclassified.' )
  }
  elsif $operatingsystem == 'Darwin' {
    warn ( 'This NTP module does not yet work on our Mac laptops.' )
  else {
    include ntp
  }
```

↑ An if statement, whose conditions are expressions that use agent-provided facts.

```
if $hostname =~ /^www(\d+)\./ {
  notify { "Welcome web server #$1": }
}
```

↑ An if statement using a regular expression and the regex match operator.

```
if 'www' in $hostname {
  ...
}
```

↑ An if statement using an `in` expression

```
case $operatingsystem {
  'Solaris':          { include role::solaris }
  'RedHat', 'CentOS': { include role::redhat  }
  /^(Debian|Ubuntu)$/:{ include role::debian  }
  default:            { include role::generic }
}
```

↑ A case statement.

```
$rootgroup = $osfamily ? {
    'Solaris'           => 'wheel',
    /(Darwin|FreeBSD)/ => 'wheel',
    default             => 'root',
}
```

↑ A selector statement being used to set the value of the `$rootgroup` variable.

```
User <| groups == 'admin' |>
```

↑ A resource collector, sometimes called the “spaceship operator.”

```
Concat::Fragment <<| tag == "bacula-storage-dir-${bacula_director}" |>>
```

↑ An exported resource collector, which works with exported resources

```
Exec {
  path        => '/usr/bin:/bin:/usr/sbin:/sbin',
  environment => 'RUBYLIB=/opt/puppet/lib/ruby/site_ruby/1.8/',
  logoutput   => true,
  timeout     => 180,
}
```

↑ A resource default for resources of the `exec` type.

```
Exec['update_migrations'] {
  environment => 'RUBYLIB=/usr/lib/ruby/site_ruby/1.8/',
}
```

↑ A resource override, which will only work in an inherited class.

```
Exec <| title == 'update_migrations' |> {
  environment => 'RUBYLIB=/usr/lib/ruby/site_ruby/1.8/',
}
```

↑ A resource override using a collector, which will work anywhere. Dangerous, but very useful in rare cases.

```
@user {'deploy':
  uid     => 2004,
  comment => 'Deployment User',
  group   => www-data,
  groups  => ["enterprise"],
  tag     => [deploy, web],
}
```

↑ A virtual resource.

```
@@nagios_service { "check_zfs${hostname}":
  use                 => 'generic-service',
  host_name           => "$fqdn",
  check_command       => 'check_nrpe_1arg!check_zfs',
  service_description => "check_zfs${hostname}",
  target              => '/etc/nagios3/conf.d/nagios_service.cfg',
  notify              => Service[$nagios::params::nagios_service],
}
```

↑ An exported resource declaration.

Language: Summary

The Puppet Language

Puppet uses its own configuration language. This language was designed to be accessible to□ sysadmins because it does not require much formal programming experience and its syntax was inspired by the Nagios configuration file format.□

The core of the Puppet language is declaring resources. Every other part of the language exists to add flexibility to the way resources are declared.

Puppet's language is mostly declarative: Rather than mandating a series of steps to carry out, a Puppet manifest simply describes a desired final state.

The resources in a manifest can be freely ordered — they will not be applied to the system in the order they are written. This is because Puppet assumes most resources aren't related to each other. If one resource depends on another, you must say so explicitly. (If you want a short section of code to get applied in the order written, you can use chaining arrows.)

Although resources can be freely ordered, several parts of the language do depend on parse order. The most notable of these are variables, which must be set before they are referenced.

Files

Puppet language files are called manifests, and are named with the `.pp` file extension. Manifest files:

- Should use UTF8 encoding
- May use Unix (LF) or Windows (CRLF) line breaks (note that the line break format also affects literal line breaks in strings)

Puppet always begins compiling with a single manifest. When using a puppet master, this file is called site.pp; when using puppet apply, it's whatever was specified on the command line.

Any classes declared in the manifest can be autoloaded from manifest files in modules. Puppet will also autoload any classes declared by an optional external node classifier.

Thus, the simplest Puppet deployment is a lone manifest file with a few resources. Complexity can grow progressively, by grouping resources into modules and classifying your nodes more granularly.

Compilation and Catalogs

Puppet manifests can use conditional logic to describe many nodes' configurations at once. Before configuring a node, Puppet compiles manifests into a catalog, which is only valid for a single node and which contains no ambiguous logic.

Catalogs are static documents which contain resources and relationships. At various stages of a Puppet run, a catalog will be in memory as a Ruby object, transmitted as JSON, and persisted to disk as YAML. The catalog format used by this version of Puppet is not documented and does not have a spec.

In the standard agent/master architecture, nodes request catalogs from a puppet master server, which compiles and serves them to nodes as needed. When running Puppet standalone with puppet

apply, catalogs are compiled locally and applied immediately.

Agent nodes cache their most recent catalog. If they request a catalog and the master fails to compile one, they will re-use their cached catalog. This recovery behavior is governed by the `usecacheonfailure` setting in puppet.conf. When testing updated manifests, you can save time by turning it off.

Example

The following short manifest manages NTP. It uses package, file, and service resources; a case statement based on a fact; variables; ordering and notification relationships; and file contents being served from a module.

```
    case $operatingsystem {
      centos, redhat: { $service_name = 'ntpd' }
      debian, ubuntu: { $service_name = 'ntp' }
    }

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

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

    file { 'ntp.conf':
      path    => '/etc/ntp.conf',
      ensure  => file,
      require => Package['ntp'],
      source  => "puppet:///modules/ntp/ntp.conf",
      # This source file would be located on the puppet master at
      # /etc/puppetlabs/puppet/modules/ntp/files/ntp.conf (in Puppet
Enterprise)
      # or
      # /etc/puppet/modules/ntp/files/ntp.conf (in open source Puppet)
    }
```

Language: Reserved Words and Acceptable Names

Reserved Words

Several words in the Puppet language are reserved. This means they:

- Cannot be used as bare word strings — you must quote these words if you wish to use them as strings.
- Cannot be used as names for custom functions.
- Cannot be used as names for classes.
- Cannot be used as names for custom resource types or defined resource types.
- Cannot be used as names of resource attributes for custom types (e.g., use “onlyif” instead of “if”)

The following words are reserved:

- `and` — expression operator
- `case` — language keyword
- `class` — language keyword
- `default` — language keyword
- `define` — language keyword
- `else` — language keyword
- `elsif` — language keyword
- `false` — boolean value
- `if` — language keyword
- `in` — expression operator
- `import` — language keyword
- `inherits` — language keyword
- `node` — language keyword
- `or` — expression operator
- `true` — boolean value
- `undef` — special value

Additionally, you cannot use the name of any existing resource type or function as the name of a function, and you cannot use the name of any existing resource type as the name of a defined type.

Reserved Class Names

The following are built-in namespaces used by Puppet and so must not be used as class names:

- `main` — Puppet automatically creates a `main` class, which contains any resources not contained by any other class.
- `settings` — The automatically created `settings` namespace contains variables with the settings available to the compiler (that is, the puppet master’s settings).

Reserved Variable Names

The following variable names are reserved, and you must not assign values to them:

- `$string` — If a variable with this name is present, all templates and inline templates in the current scope will return the value of `$string` instead of whatever they were meant to return. This is a bug rather than a deliberate design, and can be tracked at issue #14093.
- Every variable name consisting only of numbers, starting with `$0` — These regex capture variables are automatically set by regular expressions used in conditional statements, and their values do not persist outside their associated code block or selector value. Puppet's behavior when these variables are directly assigned a value is undefined.

Additionally, re-using the names of any of Puppet's built-in variables or facts at top scope will cause compilation to fail.

> Note: You can safely re-use fact names at node or local scope, but should do so with care, as dynamic scope lookup may cause classes and defined types declared in that scope to receive unexpected data.

Acceptable Characters in Names

Puppet limits the characters you can use when naming language constructs.

> Note: In some cases, names containing unsupported characters will still work. These cases should be considered bugs, and may cease to work at any time. Removal of these bug cases will not be limited to major releases.

Variables

Variable names begin with a `$` (dollar sign) and can include:

- Uppercase and lowercase letters
- Numbers
- Underscores

There is no additional restriction on the first non-$ character of a variable name. Variable names are case-sensitive. Note that some variable names are reserved.

Variable names should match the following regular expression:

```
\A\$[a-zA-Z0-9_]+\Z
```

Variable names can be fully qualified to refer to variables from foreign scopes. Qualified variable

names look like `$class::name::variable_name`. They begin with `$`, the name of the class that contains the variable, and the `::` (double colon) namespace separator, and end with the variable's local name.

Qualified variable names should match the following regular expression:

```
\A\$([a-z][a-z0-9_]*)?(::[a-z][a-z0-9_]*)*::[a-zA-Z0-9_]+\Z
```

Classes and Types

The names of classes, defined types, and custom types can consist of one or more namespace segments. Each namespace segment must begin with a lowercase letter and can include:

- Lowercase letters
- Numbers
- Underscores

Namespace segments should match the following regular expression:

```
\A[a-z][a-z0-9_]*\Z
```

The one exception is the top namespace, whose name is the empty string.

Multiple namespace segments can be joined together in a class or type name with the `::` (double colon) namespace separator.

Class names with multiple namespaces should match the following regular expression:

```
\A([a-z][a-z0-9_]*)?(::[a-z][a-z0-9_]*)*\Z
```

Note that some class names are reserved, and reserved words cannot be used as class or type names.

Modules

Module names obey the same rules as individual class/type namespace segments. That is, they must begin with a lowercase letter and can include:

- Lowercase letters
- Numbers
- Underscores

Module names should match the following regular expression:

```
\A[a-z][a-z0-9_]*\Z
```

Note that reserved words and reserved class names cannot be used as module names.

Parameters

Class and defined type parameters begin with a `$` (dollar sign), and their first non-`$` character must be a lowercase letter. They can include:

- Lowercase letters
- Numbers
- Underscores

Parameter names should match the following regular expression:

```
\A\$[a-z][a-z0-9_]*\Z
```

Tags

Tags must begin with a lowercase letter, number, or underscore, and can include:

- Lowercase letters
- Numbers
- Underscores
- Colons
- Periods
- Hyphens

Tag names should match the following regular expression:

```
\A[a-z0-9_][a-z0-9_:\.\-]*\Z
```

Resources

Resource titles may contain any characters whatsoever. They are case-sensitive.

Resource names (or namevars) may be limited by the underlying system being managed. (E.g., most systems have limits on the characters allowed in the name of a user account.) The user is generally responsible for knowing the name limits on the platforms they manage.

Nodes

The set of characters allowed in node names is undefined in this version of Puppet. For best future compatibility, you should limit node names to letters, numbers, periods, underscores, and dashes. (That is, node names should match `/\A[a-z0-9._-]+\Z/`.)

Language: Resources

- See the Type Reference for complete information about Puppet's built-in resource types.

Resources are the fundamental unit for modeling system configurations. Each resource describes some aspect of a system, like a service that must be running or a package that must be installed. The block of Puppet code that describes a resource is called a resource declaration.

Declaring a resource instructs Puppet to include it in the catalog and manage its state on the target system. Resource declarations inside a class definition or defined type are only added to the catalog once the class or an instance of the defined type is declared. Virtual resources are only added to the catalog once they are realized.

Syntax

```
# A resource declaration:
file { '/etc/passwd':
  ensure => file,
  owner  => 'root',
  group  => 'root',
  mode   => '0600',
}
```

Every resource has a type, a title, and a set of attributes:

```
type {'title':
  attribute => value,
}
```

The general form of a resource declaration is:

- The resource type, in lower-case
- An opening curly brace
- The title, which is a string
- A colon
- Optionally, any number of attribute and value pairs, each of which consists of:
  - An attribute name, which is a bare word
  - A `=>` (arrow, fat comma, or hash rocket)
  - A value, which can be any data type, depending on what the attribute requires
  - A trailing comma (note that the comma is optional after the final attribute/value pair)

- Optionally, a semicolon, followed by another title, colon, and attribute block
- A closing curly brace

Note that, in the Puppet language, whitespace is fungible.

Type

The type identifies what kind of resource it is. Puppet has a large number of built-in resource types, including files on disk, cron jobs, user accounts, services, and software packages. See here for a list of built-in resource types.

Puppet can be extended with additional resource types, written in Ruby or in the Puppet language.

Title

The title is an identifying string. It only has to identify the resource to Puppet's compiler; it does not need to bear any relationship to the actual target system.

Titles must be unique per resource type. You may have a package and a service both titled "ntp," but you may only have one service titled "ntp." Duplicate titles will cause a compilation failure.

Attributes

Attributes describe the desired state of the resource; each attribute handles some aspect of the resource.

Each resource type has its own set of available attributes; see the type reference for a complete list. Most types have a handful of crucial attributes and a larger number of optional ones. Many attributes have a default value that will be used if a value isn't specified.

Every attribute you declare must have a value; the data type of the value depends on what the attribute accepts. Most attributes that can take multiple values accept them as an array.

PARAMETERS

When discussing resources and types, parameter is a synonym for attribute. "Parameter" is usually used when discussing a type, and "attribute" is usually used when discussing an individual resource.

Behavior

A resource declaration adds a resource to the catalog, and tells Puppet to manage that resource's state. When Puppet applies the compiled catalog, it will:

- Read the actual state of the resource on the target system
- Compare the actual state to the desired state
- If necessary, change the system to enforce the desired state

Uniqueness

Puppet does not allow you to declare the same resource twice. This is to prevent multiple conflicting values from being declared for the same attribute.

Puppet uses the title and name/namevar to identify duplicate resources — if either of these is duplicated within a given resource type, the compilation will fail.

If multiple classes require the same resource, you can use a class or a virtual resource to add it to the catalog in multiple places without duplicating it.

Events

If Puppet makes any changes to a resource, it will log those changes as events. These events will appear in puppet agent's log and in the run report, which is sent to the puppet master and forwarded to any number of report processors.

Parse-Order Independence

Resources are not applied to the target system in the order they are written in the manifests — Puppet will apply the resources in whatever way is most efficient. If a resource must be applied before or after some other resource, you must explicitly say so. See Relationships for more information.

Scope Independence

Resources are not subject to scope — a resource in any scope may be referenced from any other scope, and local scopes do not introduce local namespaces for resource titles.

Containment

Resources may be contained by classes and defined types. See Containment for more details.

Special Attributes

Name/Namevar

Most types have an attribute which identifies a resource on the target system. This is referred to as the "namevar," and is often simply called "name." For example, the `name` of a service or package is the name by which the system's service or package tools will recognize it. The `path` of a file is its location on disk.

Namevar values must be unique per resource type, with only rare exceptions (such as `exec`).

Namevars are not to be confused with the title, which identifies a resource to Puppet. However, they often have the same value, since the namevar's value will default to the title if it isn't specified. Thus, the `path` of the file example above is `/etc/passwd`, even though it was never specified.

The distinction between title and namevar lets you use a single, consistently-titled resource to

manage something whose name differs by platform. For example, the NTP service is `ntpd` on Red Hat–derived systems, but `ntp` on Debian and Ubuntu; the service resource could simply be titled "ntp," but could have its name set correctly by platform. Other resources could then form relationships to it without worrying that its title will change.

Ensure

Many types have an `ensure` attribute. This generally manages the most fundamental aspect of the resource on the target system — does the file exist, is the service running or stopped, is the package installed or uninstalled, etc.

Allowed values for `ensure` vary by type. Most types accept `present` and `absent`, but there may be additional variations. Be sure to check the reference for each type you are working with.

Metaparameters

Some attributes in Puppet can be used with every resource type. These are called metaparameters. They don't map directly to system state; instead, they specify how Puppet should act toward the resource.

The most commonly used metaparameters are for specifying order relationships between resources.

You can see the full list of all metaparameters in the Metaparameter Reference.

Condensed Forms

There are two ways to compress multiple resource declarations. You can also use resource defaults to reduce duplicate typing.

Array of Titles

If you specify an array of strings as the title of a resource declaration, Puppet will treat it as multiple resource declarations with an identical block of attributes.

```
file { ['/etc',
        '/etc/rc.d',
        '/etc/rc.d/init.d',
        '/etc/rc.d/rc0.d',
        '/etc/rc.d/rc1.d',
        '/etc/rc.d/rc2.d',
        '/etc/rc.d/rc3.d',
        '/etc/rc.d/rc4.d',
        '/etc/rc.d/rc5.d',
        '/etc/rc.d/rc6.d']:
  ensure => directory,
  owner  => 'root',
  group  => 'root',
  mode   => 0755,
}
```

This example is the same as declaring each directory as a separate resource with the same attribute block. You can also store an array in a variable and specify the variable as a resource title:

```
$rcdirectories = ['/etc',
                  '/etc/rc.d',
                  '/etc/rc.d/init.d',
                  '/etc/rc.d/rc0.d',
                  '/etc/rc.d/rc1.d',
                  '/etc/rc.d/rc2.d',
                  '/etc/rc.d/rc3.d',
                  '/etc/rc.d/rc4.d',
                  '/etc/rc.d/rc5.d',
                  '/etc/rc.d/rc6.d']

file { $rcdirectories:
  ensure => directory,
  owner  => 'root',
  group  => 'root',
  mode   => 0755,
}
```

Note that you cannot specify a separate namevar with an array of titles, since it would then be duplicated across all of the resources. Thus, each title must be a valid namevar value.

Semicolon After Attribute Block

If you end an attribute block with a semicolon rather than a comma, you may specify another title, another colon, and another complete attribute block, instead of closing the curly braces. Puppet will treat this as multiple resources of a single type.

```
file {
  '/etc/rc.d':
    ensure => directory,
    owner  => 'root',
    group  => 'root',
    mode   => 0755;

  '/etc/rc.d/init.d':
    ensure => directory,
    owner  => 'root',
    group  => 'root',
    mode   => 0755;

  '/etc/rc.d/rc0.d':
    ensure => directory,
    owner  => 'root',
    group  => 'root',
    mode   => 0755;
}
```

Adding or Modifying Attributes

Although you cannot declare the same resource twice, you can add attributes to an already-declared resource. In certain circumstances, you can also override attributes.

Amending Attributes With a Reference

```
file {'/etc/passwd':
  ensure => file,
}

File['/etc/passwd'] {
  owner => 'root',
  group => 'root',
  mode  => 0640,
}
```

The general form of a reference attribute block is:

- A reference to the resource in question (or a multi-resource reference)
- An opening curly brace
- Any number of attribute => value pairs
- A closing curly brace

In normal circumstances, this idiom can only be used to add previously unmanaged attributes to a resource; it cannot override already-specified attributes. However, within an inherited class, you can use this idiom to override attributes.

Amending Attributes With a Collector

```
class base::linux {
  file {'/etc/passwd':
    ensure => file,
  }
  ...
}

include base::linux

File <| tag == 'base::linux' |> {
  owner => 'root',
  group => 'root',
  mode  => 0640,
}
```

The general form of a collector attribute block is:

- A resource collector that matches any number of resources

- An opening curly brace
- Any number of attribute => value (or attribute +> value) pairs
- A closing curly brace

Much like in an inherited class, you can use the special `+>` keyword to append values to attributes that accept arrays. See appending to attributes for more details.

Note that this idiom must be used carefully, if at all:

- It can always override already-specified attributes, regardless of class inheritance.
- It can affect large numbers of resources at once.
- It will implicitly realize any virtual resources that the collector matches. If you are using virtual resources at all, you must use extreme care when constructing collectors that are not intended to realize resources, and would be better off avoiding non-realizing collectors entirely.
- Since it ignores class inheritance, you can override the same attribute twice, which results in a parse-order dependent race where the final override wins.

Language: Relationships and Ordering

The order of resources in a Puppet manifest does not matter. Puppet assumes that most resources are not related to each other and will manage the resources in whatever order is most efficient.

If a group of resources should be managed in a specific order, you must explicitly declare the relationships.

Syntax

Relationships can be declared with the relationship metaparameters, chaining arrows, and the `require` function.

Relationship Metaparameters

```
package { 'openssh-server':
  ensure => present,
  before => File['/etc/ssh/sshd_config'],
}
```

Puppet uses four metaparameters to establish relationships. Each of them can be set as an attribute in any resource. The value of any relationship metaparameter should be a resource reference (or array of references) pointing to one or more target resources.

`before`

> Causes a resource to be applied before the target resource.

`require`

> Causes a resource to be applied after the target resource.

`notify`

> Causes a resource to be applied before the target resource. The target resource will refresh if the notifying resource changes.

`subscribe`

> Causes a resource to be applied after the target resource. The subscribing resource will refresh if the target resource changes.

If two resources need to happen in order, you can either put a `before` attribute in the prior one or a `require` attribute in the subsequent one; either approach will create the same relationship. The same is true of `notify` and `subscribe`.

The two examples below create the same ordering relationship:

```
package { 'openssh-server':
  ensure => present,
  before => File['/etc/ssh/sshd_config'],
}
```

```
file { '/etc/ssh/sshd_config':
  ensure  => file,
  mode    => 600,
  source  => 'puppet:///modules/sshd/sshd_config',
  require => Package['openssh-server'],
}
```

The two examples below create the same notification relationship:

```
file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => 600,
  source => 'puppet:///modules/sshd/sshd_config',
  notify => Service['sshd'],
}
```

```
service { 'sshd':
  ensure    => running,
  enable    => true,
```

```
      subscribe => File['/etc/ssh/sshd_config'],
    }
```

Chaining Arrows

```
    # ntp.conf is applied first, and will notify the ntpd service if it
changes:
    File['/etc/ntp.conf'] ~> Service['ntpd']
```

You can create relationships between two resources or groups of resources using the `->` and `~>` operators.

`->` **(ordering arrow)**

> Causes the resource on the left to be applied before the resource on the right. Written with a hyphen and a greater-than sign.

`~>` **(notification arrow)**

> Causes the resource on the left to be applied first, and sends a refresh event to the resource on the right if the left resource changes. Written with a tilde and a greater-than sign.

OPERANDS

The chaining arrows accept the following types of operands on either side of the arrow:

- Resource references, including multi-resource references
- Resource declarations
- Resource collectors

> Note: Arrays of references cannot be chained. To chain multiple resources at once, you must use a multi-resource reference or a collector.

An operand can be shared between two chaining statements, which allows you to link them together into a "timeline:"

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

Since resource declarations can be chained, you can use chaining arrows to make Puppet apply a section of code in the order that it is written:

```
    # first:
    package { 'openssh-server':
      ensure => present,
    } -> # and then:
    file { '/etc/ssh/sshd_config':
```

```
    ensure => file,
    mode   => 600,
    source => 'puppet:///modules/sshd/sshd_config',
  } ~> # and then:
  service { 'sshd':
    ensure => running,
    enable => true,
  }
```

And since collectors can be chained, you can create many-to-many relationships:

```
Yumrepo <| |> -> Package <| |>
```

This example would apply all yum repository resources before applying any package resources, which protects any packages that rely on custom repos.

> Note: Chained collectors can potentially cause huge dependency cycles and should be used carefully. They can also be dangerous when used with virtual resources, which are implicitly realized by collectors.

> Note: Collectors can only search on attributes which are present in the manifests and cannot see properties that must be read from the target system. For example, if the example above had been written as `Yumrepo <| |> -> Package <| provider == yum |>`, it would only create relationships with packages whose `provider` attribute had been explicitly set to `yum` in the manifests. It would not affect any packages that didn't specify a provider but would end up using Yum.

REVERSED FORMS

Both chaining arrows have a reversed form (`<-` and `<~`). As implied by their shape, these forms operate in reverse, causing the resource on their right to be applied before the resource on their left.

> Note: As the majority of Puppet's syntax is written left-to-right, these reversed forms can be confusing and are not recommended.

The `require` Function

The `require` function declares a class and causes it to become a dependency of the surrounding container:

```
class wordpress {
  require apache
```

```
    require mysql
    ...
  }
```

The above example would cause every resource in the `apache` and `mysql` classes to be applied before any of the resources in the `wordpress` class.

Unlike the relationship metaparameters and chaining arrows, the `require` function does not have a reciprocal form or a notifying form. However, more complex behavior can be obtained by combining `include` and chaining arrows inside a class definition:

```
  class apache::ssl {
    include site::certificates
    # Restart every service in this class if any of our SSL certificates
change on disk:
    Class['site::certificates'] ~> Class['apache::ssl']
  }
```

Behavior

Ordering and Notification

Puppet has two types of resource relationships:

- Ordering
- Ordering with notification

An ordering relationship ensures that one resource will be managed before another.

A notification relationship does the same, but also sends the latter resource a refresh event if Puppet changes the first resource's state. A refresh event causes the recipient to refresh itself.

Refreshing

Only certain resource types can refresh themselves. Of the built-in types, these are service, mount, and exec.

Service resources refresh by restarting their service. Mount resources refresh by unmounting and then mounting their volume. Exec resources usually do not refresh, but can be made to: setting `refreshonly => true` causes the exec to never fire unless it receives a refresh event. You can also set an additional `refresh` command, which causes the exec to run both commands when it receives a refresh event.

Parse-Order Independence

Relationships are not limited by parse-order. You can declare a relationship with a resource before that resource has been declared.

Missing Dependencies

If one of the resources in a relationship is never declared, compilation will fail with one of the following errors:

- `Could not find dependency <OTHER RESOURCE> for <RESOURCE>`
- `Could not find resource '<OTHER RESOURCE>' for relationship on '<RESOURCE>'`.

Failed Dependencies

If Puppet fails to apply the prior resource in a relationship, it will skip the subsequent resource and log the following messages:

```
notice: <RESOURCE>: Dependency <OTHER RESOURCE> has failures: true
warning: <RESOURCE>: Skipping because of failed dependencies
```

It will then continue to apply any unrelated resources. Any resources that depend on the skipped resource will also be skipped.

This helps prevent inconsistent system state by causing a “clean” failure instead of attempting to apply a resource whose prerequisites may be broken.

> Note: Although a resource won’t be applied if a dependency fails, it can still receive and respond to refresh events from other, successful, dependencies. This is because refreshes are handled semi-independently of the normal resource sync process. It is an outstanding design issue, and may be tracked at issue #7486.

Dependency Cycles

If two or more resources require each other in a loop, Puppet will compile the catalog but will be unable to apply it. Puppet will log an error like the following, and will attempt to help you identify the cycle:

```
err: Could not apply complete catalog: Found 1 dependency cycle:
(<RESOURCE> => <OTHER RESOURCE> => <RESOURCE>)
Try the '--graph' option and opening the resulting '.dot' file in OmniGraffle
or GraphViz
```

To locate the directory containing the graph files, run `puppet agent --configprint graphdir`.

Language: Resource Defaults

Resource defaults let you set default attribute values for a given resource type. Any resource

declaration within the area of effect that omits those attributes will inherit the default values.

Syntax

```
Exec {
  path        => '/usr/bin:/bin:/usr/sbin:/sbin',
  environment => 'RUBYLIB=/opt/puppet/lib/ruby/site_ruby/1.8/',
  logoutput   => true,
  timeout     => 180,
}
```

The general form of resource defaults is:

- The resource type, capitalized. (If the type has a namespace separator (`::`) in its name, every segment must be capitalized. E.g., `Concat::Fragment`.)
- An opening curly brace.
- Any number of attribute and value pairs.
- A closing curly brace.

You can specify defaults for any resource type in Puppet, including defined types.

Behavior

Within the area of effect, every resource of the specified type that omits a given attribute will inherit that attribute's default value.

Attributes that are set explicitly in a resource declaration will always override any default value.

Resource defaults are parse-order independent. A default will affect resource declarations written both above and below it.

Overriding Defaults From Parent Scopes

Resource defaults declared in the local scope will override any defaults received from parent scopes.

Overriding of resource defaults is per attribute, not per block of attributes. Thus, local and inherited resource defaults that don't conflict with each other will be merged together.

Area of Effect

Resource defaults obey dynamic scope; see here for a full description of scope rules.

You can declare global resource defaults in the site manifest outside any node definition.

Unlike dynamic variable lookup, dynamic scope for resource defaults is not deprecated in Puppet 2.7.

Language: Variables

Syntax

Assignment

```
$content = "some content\n"
```

Variable names are prefixed with a `$` (dollar sign). Values are assigned to them with the `=` (equal sign) assignment operator.

Any value of any of the normal (i.e. non-regex) data types can be assigned to a variable. Any statement that resolves to a normal value (including expressions, functions, and other variables) can be used in place of a literal value. The variable will contain the value that the statement resolves to, rather than a reference to the statement.

Variables can only be assigned using their short name. That is, a given scope cannot assign values to variables in a foreign scope.

Resolution

```
file {'/tmp/testing':
  ensure => file,
  content => $content,
}

$address_array = [$address1, $address2, $address3]
```

The name of a variable can be used in any place where a value of its data type would be accepted, including expressions, functions, and resource attributes. Puppet will replace the name of the variable with its value.

Interpolation

```
$rule = "Allow * from $ipaddress"
file { "${homedir}/.vim":
  ensure => directory,
  ...
}
```

Puppet can resolve variables in double-quoted strings; this is called “interpolation.”

Inside a double-quoted string, you can optionally surround the name of the variable (the portion after the `$`) with curly braces (`${var_name}`). This syntax helps to avoid ambiguity and allows

variables to be placed directly next to non-whitespace characters. These optional curly braces are only allowed inside strings.

Appending Assignment

When creating a local variable with the same name as a variable in top scope, node scope, or a parent scope, you can optionally append to the received value with the `+=` (plus-equals) appending assignment operator.

```
$ssh_users = ['myself', 'someone']

class test {
  $ssh_users += ['someone_else']
}
```

In the example above, the value of `$ssh_users` inside class `test` would be `['myself', 'someone', 'someone_else']`.

The value appended with the `+=` operator must be the same data type as the received value. This operator can only be used with strings, arrays, and hashes:

- Strings: Will concatenate the two strings.
- Arrays: Will add the elements of the appended array to the end of the received array.
- Hashes: Will merge the two hashes.

Behavior

Scope

The area of code where a given variable is visible is dictated by its scope. Variables in a given scope are only available within that scope and its child scopes, and any local scope can locally override the variables it receives from its parents.

See the section on scope for complete details.

Accessing Out-of-Scope Variables

You can access out-of-scope variables from named scopes by using their qualified names:

```
$vhostdir = $apache::params::vhostdir
```

Note that the top scope's name is the empty string. See scope for details.

No Reassignment

Unlike most other languages, Puppet only allows a given variable to be assigned once within a given scope. You may not change the value of a variable, although you may assign a different value

to the same variable name in a new scope:

```
# scope-example.pp
# Run with puppet apply --certname www1.example.com scope-example.pp
$myvar = "Top scope value"
node 'www1.example.com' {
  $myvar = "Node scope value"
  notice( "from www1: $myvar" )
  include myclass
}
node 'db1.example.com' {
  notice( "from db1: $myvar" )
  include myclass
}
class myclass {
  $myvar = "Local scope value"
  notice( "from myclass: $myvar" )
}
```

In the example above, `$myvar` has several different values, but only one value will apply to any given scope.

> Note: Due to insufficient protection of the scope object that gets passed into templates, it is possible to reassign a variable inside a template and have the new value persist in the Puppet scope after the template is evaluated. This behavior is considered a bug; do not use it. It will not be removed during the Puppet 2.7 series, but may be removed thereafter without a deprecation period.

Parse-Order Dependence

Unlike resource declarations, variable assignments are parse-order dependent. This means you cannot resolve a variable before it has been assigned.

This is the main way in which the Puppet language fails to be fully declarative.

Naming

Variable names are case-sensitive and can include alphanumeric characters and underscores.

Qualified variable names are prefixed with the name of their scope and the `::` (double colon) namespace separator. (For example, the `$vhostdir` variable from the `apache::params` class would be `$apache::params::vhostdir`.)

See the section on acceptable characters in variable names for more details. Additionally, several variable names are reserved.

Facts and Built-In Variables

Puppet provides several built-in top-scope variables, which you can rely on in your own manifests.

Facts

Each node submits a very large number of facts (as discovered by Facter) when requesting its catalog, and all of them are available as top-scope variables in your manifests. In addition to the built-in facts, you can create and distribute custom facts as plugins.

- See here for a complete list of built-in facts.
- See here for a guide to writing custom facts.
- Run `facter -p` on one of your nodes to get a complete report of the facts that node will report to the master.

Agent-Set Variables

Puppet agent sets several additional variables for a node which are available when compiling that node's catalog:

- `$environment` — the node's current environment.
- `$clientcert` — the node's certname setting.
- `$clientversion` — the current version of puppet agent.

Master-Set Variables

These variables are set by the puppet master and are most useful when managing Puppet with Puppet. (For example, managing puppet.conf with a template.)

- `$servername` — the puppet master's fully-qualified domain name. (Note that this information is gathered from the puppet master by Facter, rather than read from the config files; even if the master's certname is set to something other than its fully-qualified domain name, this variable will still contain the server's fqdn.)
- `$serverip` — the puppet master's IP address.
- `$serverversion` — the current version of puppet on the puppet master.
- `$settings::<name of setting>` — the value of any of the master's configuration settings. This is implemented as a special namespace and these variables must be referred to by their qualified names. Note that, other than `$environment`, the agent node's settings are not available in manifests. If you wish to expose them to the master in this version of Puppet (2.7), you will have to create a custom fact.

Parser-Set Variables

These variables are set in every local scope by the parser during compilation. These are mostly useful when implementing complex defined types.

- `$module_name` — the name of the module that contains the current class or defined type.

- `$caller_module_name` — the name of the module in which the specific instance of the surrounding defined type was declared. This is only useful when creating versatile defined types which will be re-used by several modules.

Language: Scope

Scope Basics

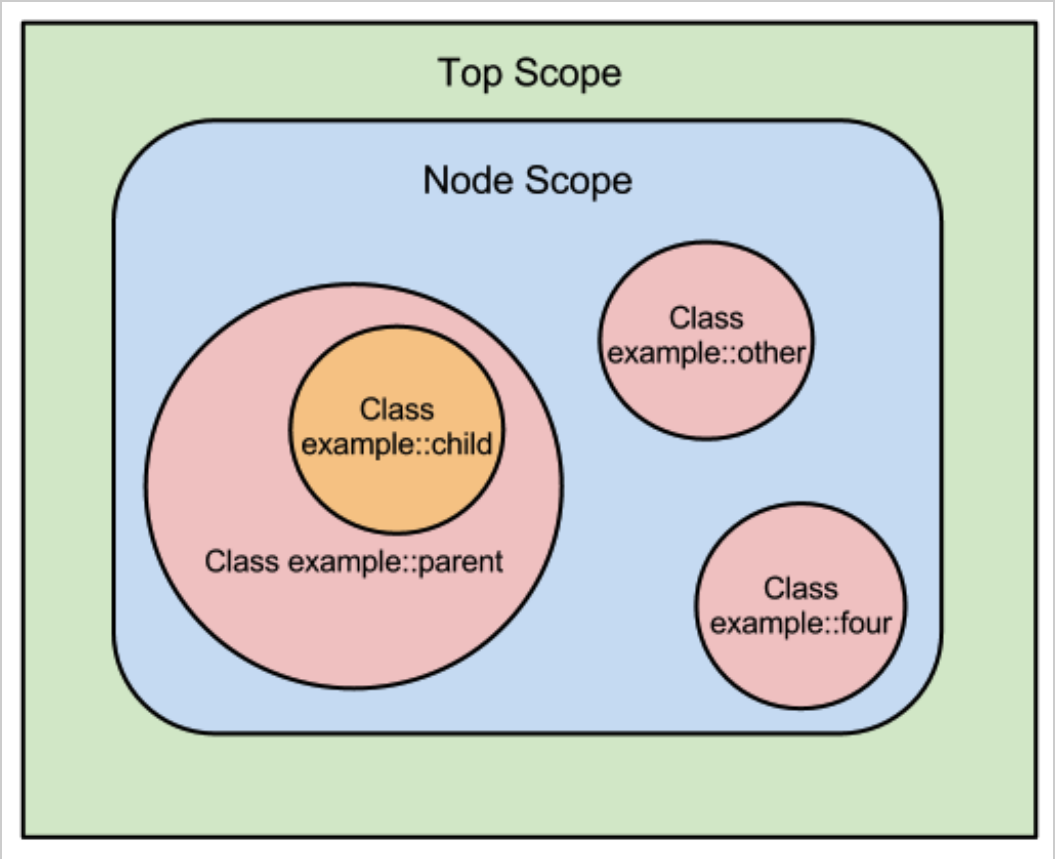
A scope is a specific area of code, which is partially isolated from other areas of code. Scopes limit the reach of:

- Variables
- Resource defaults

Scopes do not limit the reach of:

- Resource titles, which are all global
- Resource references, which can refer to a resource declared in any scope

Summary Diagram



Any given scope has access to its own contents, and also receives additional contents from its parent scope, from node scope, and from top scope.

In the diagram above:

- Top scope can only access variables and defaults from its own scope.
- Node scope can access variables and defaults from its own scope and top scope.
- Each of the `example::parent, example::other`, and `example::four` classes can access variables and defaults from their own scope, node scope, and top scope.
- The `example::child` class can access variables and defaults from its own scope, `example::parent`'s scope, node scope, and top scope.

Top Scope

Code that is outside any class definition, type definition, or node definition exists at top scope. Variables and defaults declared at top scope are available everywhere.

```
# site.pp
$variable = "Hi!"

class example {
  notify {"Message from elsewhere: $variable":}
}

include example
```

```
$ puppet apply site.pp
notice: Message from elsewhere: Hi!
```

Node Scope

Code inside a node definition exists at node scope. Note that since only one node definition can match a given node, only one node scope can exist at a time.

Variables and defaults declared at node scope are available everywhere except top scope.

> Note: Classes and resources declared at top scope bypass node scope entirely, and so cannot access variables or defaults from node scope.

```
# site.pp
$top_variable = "Available!"
node 'puppet.example.com' {
  $variable = "Hi!"
  notify {"Message from here: $variable":}
  notify {"Top scope: $top_variable":}
}
notify {"Message from top scope: $variable":}
```

```
$ puppet apply site.pp
notice: Message from here: Hi!
notice: Top scope: Available!
notice: Message from top scope:
```

In this example, node scope can access top scope variables, but not vice-versa.

Local Scopes

Code inside a class definition or defined type exists in a local scope.

Variables and defaults declared in a local scope are only available in that scope and its children. There are two different sets of rules for when scopes are considered related; see "scope lookup rules" below.

```
# /etc/puppet/modules/scope_example/manifests/init.pp
class scope_example {
  $variable = "Hi!"
  notify {"Message from here: $variable":}
  notify {"Node scope: $node_variable Top scope: $top_variable":}
}

# /etc/puppet/manifests/site.pp
$top_variable = "Available!"
node 'puppet.example.com' {
  $node_variable = "Available!"
  include scope_example
  notify {"Message from node scope: $variable":}
}
notify {"Message from top scope: $variable":}
```

```
$ puppet apply site.pp
notice: Message from here: Hi!
notice: Node scope: Available! Top scope: Available!
notice: Message from node scope:
notice: Message from top scope:
```

In this example, a local scope can see "out" into node and top scope, but outer scopes cannot see "in."

Overriding Received Values

Variables and defaults declared at node scope can override those received from top scope. Those declared at local scope can override those received from node and top scope, as well as any parent scopes. That is: if multiple variables with the same name are available, Puppet will use the "most local" one.

```
# /etc/puppet/modules/scope_example/manifests/init.pp
class scope_example {
```

```
    $variable = "Hi, I'm local!"
    notify {"Message from here: $variable":}
  }

  # /etc/puppet/manifests/site.pp
  $variable = "Hi, I'm top!"

  node 'puppet.example.com' {
    $variable = "Hi, I'm node!"
    include scope_example
  }
```

```
$ puppet apply site.pp
notice: Message from here: Hi, I'm local!
```

Resource defaults are processed by attribute rather than as a block. Thus, defaults that declare different attributes will be merged, and only the attributes that conflict will be overridden.

```
  # /etc/puppet/modules/scope_example/manifests/init.pp
  class scope_example {
    File { ensure => directory, }

    file {'/tmp/example':}
  }

  # /etc/puppet/manifests/site.pp
  File {
    ensure => file,
    owner  => 'puppet',
  }

  include scope_example
```

In this example, `/tmp/example` would be a directory owned by the `puppet` user, and would combine the defaults from top and local scope.

More Details

Scope of External Node Classifier Data

- Variables provided by an ENC are set at top scope.
- However, all of the classes assigned by an ENC are declared at node scope.

This gives approximately the best and most-expected behavior — variables from an ENC are available everywhere, and classes may use node-specific variables.

> Note: this means compilation will fail if an ENC tries to set a variable that is already set at top scope by the site manifest.

Named Scopes and Anonymous Scopes

A class definition creates a named scope, whose name is the same as the class's name. Top scope is also a named scope; its name is the empty string (aka, the null string).

Node scope and the local scopes created by defined resources are anonymous and cannot be directly referenced.

Accessing Out-of-Scope Variables

Variables declared in named scopes can be referenced directly from anywhere (including scopes that otherwise would not have access to them) by using their global qualified name.

Qualified variable names are formatted as follows, using the double-colon namespace separator between segments:

```
$<NAME OF SCOPE>::<NAME OF VARIABLE>
```

```
include apache::params
$local_copy = $apache::params::confdir
```

This example would set the variable `$local_copy` to the value of the `$confdir` variable from the `apache::params` class.

Notes:

- Remember that top scope's name is the empty string (a.k.a, the null string). Thus, `$::my_variable` would always refer to the top-scope value of `$my_variable`, even if `$my_variable` has a different value in local scope.
- Note that a class must be declared in order to access its variables; simply having the class available in your modules is insufficient.

  This means the availability of out-of-scope variables is parse order dependent. You should only access out-of-scope variables if the class accessing them can guarantee that the other class is already declared, usually by explicitly declaring it with `include` before trying to read its variables.

Variables declared in anonymous scopes can only be accessed normally and do not have global qualified names.

Scope Lookup Rules

The scope lookup rules determine when a local scope becomes the parent of another local scope.

There are two different sets of scope lookup rules: static scope and dynamic scope. Puppet 2.7 uses dynamic scope, but future versions of Puppet will use static scope. To help users prepare, Puppet 2.7 will print warnings to its log file whenever a variable's value would be different under static scope.

More details about the elimination of dynamic scope can be found here.

Static Scope

In static scope, parent scopes are only assigned by class inheritance (using the `inherits` keyword). Any derived class receives the contents of its base class in addition to the contents of node and top scope.

All other local scopes have no parents — they only receive their own contents, and the contents of node scope (if applicable) and top scope.

Static scope has the following characteristics:

- Scope contents are predictable and do not depend on parse order.
- Scope contents can be determined simply by looking at the relevant class definition(s); the place where a class or type is declared has no effect. (The only exception is node definitions — if a class is declared outside a node, it does not receive the contents of node scope.)

All future versions of Puppet will use static scope for looking up variables. Puppet 2.7 does not enforce static scope, but will log warnings when a variable lookup would violate it.

Dynamic Scope

In dynamic scope, parent scopes are assigned by both inheritance and declaration, with preference being given to inheritance. The full list of rules is:

- Each scope has only one parent, but may have an unlimited chain of grandparents, and receives the merged contents of all of them (with nearer ancestors overriding more distant ones).
- The parent of a derived class is its base class.
- The parent of any other class or defined resource is the first scope in which it was declared.
- When you declare a derived class whose base class hasn't already been declared, the base class is immediately declared in the current scope, and its parent assigned accordingly. This effectively "inserts" the base class between the derived class and the current scope. (If the base class has already been declared elsewhere, its existing parent scope is not changed.)

Dynamic scope has the following characteristics:

- A scope's parent cannot be identified by looking at the definition of a class — you must

examine every place where the class or resource may have been declared.

- In some cases, you can only determine a scope's contents by executing the code.
- Since classes may be declared multiple times with the `include` function, the contents of a given scope are parse-order dependent, and may vary between different runs of identical code. This is less of a danger in Puppet 2.7 than in previous versions, as relative resource ordering is now deterministic, but can still cause problems when running on similar-but-not-identical nodes.

If used simply, dynamic scope will usually yield simple results; in fact, it can emulate static scope. However, combining classes that declare classes, class inheritance, and insufficiently unique variable names can result in erratic behavior. Adding node inheritance to that mix can make the confusion an order of magnitude worse. See "Scope and Puppet" for historical context and for better solutions to deal with per-node data.

Messy Under-the-Hood Details

- Node scope only exists if there is at least one node definition in the site manifest (or one has been imported into it). If no node definitions exist, then ENC classes get declared at top scope.
- Although top scope and node scope are described above as being special scopes, they are actually implemented as part of the chain of parent scopes, with node scope being a child of top scope and the parent of any classes declared inside the node definition. However, since the move to static scoping causes them to behave as little islands of dynamic scoping in a statically scoped world, it's simpler to think of them as special cases.
- If you ignore best practices and use node inheritance, the rules of parent scope assignment treat node definitions like classes; that is, the base node becomes the parent scope of the derived node, and normal dynamic scoping will apply to the classes declared in each of the two definitions. Note that this will usually yield the opposite result of whatever you are trying to achieve.

Language: Conditional Statements

Conditional statements let your Puppet code behave differently in different situations. They are most helpful when combined with facts or with data retrieved from an external source.

Summary

Puppet 2.7 supports "if" statements, case statements, and selectors.

An "if" statement:

```
    if $is_virtual == 'true' {
      warning('Tried to include class ntp on virtual machine; this node may be
misclassified.')
```

```
    }
    elsif $operatingsystem == 'Darwin' {
      warning('This NTP module does not yet work on our Mac laptops.')
    }
    else {
      include ntp
    }
```

A case statement:

```
    case $operatingsystem {
      'Solaris':          { include role::solaris }
      'RedHat', 'CentOS': { include role::redhat  }
      /^(Debian|Ubuntu)$/:{ include role::debian  }
      default:            { include role::generic }
    }
```

A selector:

```
    $rootgroup = $osfamily ? {
        'Solaris'          => 'wheel',
        /(Darwin|FreeBSD)/ => 'wheel',
        default            => 'root',
    }

    file { '/etc/passwd':
      ensure => file,
      owner  => 'root',
      group  => $rootgroup,
    }
```

"If" Statements

"If" statements take a boolean condition and an arbitrary block of Puppet code, and will only execute the block if the condition is true. They can optionally include `elsif` and `else` clauses.

Syntax

```
    if $is_virtual == 'true' {
      # Our NTP module is not supported on virtual machines:
      warn( 'Tried to include class ntp on virtual machine; this node may be
misclassified.' )
    }
    elsif $operatingsystem == 'Darwin' {
      warn ( 'This NTP module does not yet work on our Mac laptops.' )
    }
    else {
      # Normal node, include the class.
      include ntp
    }
```

The general form of an "if" statement is:

- The `if` keyword
- A condition
- A pair of curly braces containing any Puppet code
- Optionally: the `elsif` keyword, another condition, and a pair of curly braces containing Puppet code
- Optionally: the `else` keyword and a pair of curly braces containing Puppet code

Behavior

Puppet's "if" statements behave much like those in any other language. The `if` condition is processed first and, if it is true, only the `if` code block is executed. If it is false, each `elsif` condition (if present) is tested in order, and if all conditions fail, the `else` code block (if present) is executed.

If none of the conditions in the statement match and there is no `else` block, Puppet will do nothing and move on.

"If" statements will execute a maximum of one code block.

Conditions

The condition(s) of an "if" statement may be any fragment of Puppet code that resolves to a boolean value. This includes:

- Variables
- Expressions, including arbitrarily nested `and` and `or` expressions
- Functions that return values

Fragments that resolve to non-boolean values will be automatically converted to booleans as described here.

Static values may also be conditions, although doing this would be pointless.

REGEX CAPTURE VARIABLES

If you use the regular expression match operator in a condition, any captures from parentheses in the pattern will be available inside the associated code block as numbered variables (`$1, $2`, etc.), and the entire match will be available as `$0`:

```
if $hostname =~ /^www(\d+)\./ {
  notice("Welcome to web server number $1")
}
```

This example would capture any digits from a hostname like `www01` and `www02` and store them in the `$1` variable.

These are not normal variables, and have some special behaviors:

- The values of the numbered variables do not persist outside the code block associated with the pattern that set them.
- In nested conditionals, each conditional has its own set of values for the set of numbered variables. At the end of an interior statement, the numbered variables are reset to their previous values for the remainder of the outside statement. (This causes conditional statements to act like local scopes, but only with regard to the numbered variables.)

Case Statements

Like "if" statements, case statements choose one of several blocks of arbitrary Puppet code to execute. They take a control expression and a list of cases and code blocks, and will execute the first block whose case value matches the control expression.

Syntax

```
case $operatingsystem {
  'Solaris':          { include role::solaris } # apply the solaris class
  'RedHat', 'CentOS': { include role::redhat  } # apply the redhat class
  /^(Debian|Ubuntu)$/:{ include role::debian  } # apply the debian class
  default:            { include role::generic } # apply the generic class
}
```

The general form of a case statement is:

- The `case` keyword
- A control expression (see below)
- An opening curly brace
- Any number of possible matches, which consist of:
  - A case (see below) or comma-separated list of cases
  - A colon
  - A pair of curly braces containing any arbitrary Puppet code
- A closing curly brace

Behavior

Puppet compares the control expression to each of the cases, in the order they are listed. It will execute the block of code associated with the first matching case, and ignore the remainder of the statement.

- Basic cases are compared with the `==` operator (which is case-insensitive).

- Regular expression cases are compared with the `=~` operator (which is case-sensitive).
- The special `default` case matches anything.

If none of the cases match, Puppet will do nothing and move on.

Case statements will execute a maximum of one code block.

Control Expressions

The control expression of a case statement may be any fragment of Puppet code that resolves to a normal value. This includes:

- Variables
- Expressions
- Functions that return values

Cases

Cases may be any of the following:

- A literal value (remember to quote strings)
- A variable
- A function call that returns a value
- A regular expression
- The special bare word value `default`

Note that you cannot use arbitrary expressions or selectors as cases.

You may use a comma-separated list of cases to associate more than one case with the same block of code.

Normal values are compared to the control expression using the `==` operator, and regular expressions are compared with the `=~` operator. The special `default` case matches any control expression.

Cases are compared in the order that they are written in the manifest; thus, the `default` case (if any) must be at the end of the list.

REGEX CAPTURE VARIABLES

If you use regular expression cases, any captures from parentheses in the pattern will be available inside the associated code block as numbered variables (`$1, $2`, etc.), and the entire match will be available as `$0`:

```
if $hostname =~ /^www(\d+)\./ {
  notice("Welcome to web server number $1")
}
```

This example would capture any digits from a hostname like `www01` and `www02` and store them in the `$1` variable.

These are not normal variables, and have some special behaviors:

- The values of the numbered variables do not persist outside the code block associated with the pattern that set them.
- In nested conditionals, each conditional has its own set of values for the set of numbered variables. At the end of an interior statement, the numbered variables are reset to their previous values for the remainder of the outside statement. (This causes conditional statements to act like local scopes, but only with regard to the numbered variables.)

ASIDE: BEST PRACTICES

Case statements should usually have a default case.

- If the rest of your cases are meant to be comprehensive, putting a `fail('message')` call in the default case makes your code more robust by protecting against mystery failures due to behavior changes elsewhere in your manifests.
- If your cases aren't comprehensive and nodes that match none should do nothing, write a default case with an empty code block (`default: {}`). This makes your intention obvious to the next person who has to maintain your code.

Selectors

Selector statements are similar to case statements, but return a value instead of executing a code block.

Location

Selectors must be used at places in the code where a plain value is expected. This includes:

- Variable assignments
- Resource attributes
- Function arguments
- Resource titles
- A value in another selector
- Expressions

Selectors are not legal in:

- A case in another selector
- A case in a case statement

ASIDE: BEST PRACTICES

For readability's sake, you should generally only use selectors in variable assignments.

Syntax

Selectors resemble a cross between a case statement and the ternary operator found in other languages.

```
$rootgroup = $osfamily ? {
    'Solaris'           => 'wheel',
    /(Darwin|FreeBSD)/ => 'wheel',
    default             => 'root',
}

file { '/etc/passwd':
  ensure => file,
  owner  => 'root',
  group  => $rootgroup,
}
```

In the example above, the value of `$rootgroup` is determined using the value of `$osfamily`.

The general form of a selector is:

- A control variable
- The `?` (question mark) keyword
- An opening curly brace
- Any number of possible matches, each of which consists of:
  - A case
  - The `=>` (fat comma) keyword
  - A value
  - A trailing comma
- A closing curly brace

Behavior

The entire selector statement is treated as a single value.

Puppet compares the control variable to each of the cases, in the order they are listed. When it finds a matching case, it will treat that value as the value of the statement and ignore the remainder of the statement.

- Basic cases are compared with the `==` operator (which is case-insensitive).

- Regular expression cases are compared with the `=~` operator (which is case-sensitive).
- The special `default` case matches anything.

If none of the cases match, Puppet will fail compilation with a parse error. Consequently, a default case should be considered mandatory.

Control Variables

Control variables in selectors must be variables or functions that return values. You cannot use expressions as control variables.

Cases

Cases may be any of the following:

- A literal value (remember to quote strings)
- A variable
- A function call that returns a value
- A regular expression
- The special bare word value `default`

Note that you cannot use arbitrary expressions or selectors as cases.

Unlike in case statements, you cannot use lists of cases. If you need more than one case associated with a single value, you must use a regular expression.

Normal values are compared to the control variable using the `==` operator, and regular expressions are compared with the `=~` operator. The special `default` case matches any control variable.

Cases are compared in the order that they are written in the manifest; thus, the `default` case (if any) must be at the end of the list.

REGEX CAPTURE VARIABLES

If you use regular expression cases, any captures from parentheses in the pattern will be available inside the associated value as numbered variables (`$1, $2`, etc.), and the entire match will be available as `$0`:

```
$system = $operatingsystem ? {
  /(RedHat|Debian)/ => "our system is $1",
  default           => "our system is unknown",
}
```

These are not normal variables, and have some special behaviors:

- The values of the numbered variables do not persist outside the value associated with the pattern that set them.

- In nested conditionals, each conditional has its own set of values for the set of numbered variables. At the end of an interior statement, the numbered variables are reset to their previous values for the remainder of the outside statement. (This causes conditional statements to act like local scopes, but only with regard to the numbered variables.)

Values

Values may be any of the following:

- Any literal value, with the exception of hash literals
- A variable
- A function call that returns a value
- Another selector

Note that you cannot use arbitrary expressions as values.

Language: Expressions

Expressions resolve to values and can be used in most of the places where values of the standard data types are required. Expressions can be compounded with other expressions and the entire combined expression will resolve to a single value.

Most expressions resolve to boolean values. They are particularly useful as conditions in conditional statements.

Location

Expressions can be used in the following places:

- The operand of another expression
- The condition of an if statement
- The control expression of a case statement
- The assignment of a variable
- The value of a resource attribute
- The argument(s) of a function call

They cannot be used in selectors or as resource titles.

Syntax

An expression consists of two operands separated by an operator; the only operator that takes one operand is `!` (not).

```
5 < 9
```

```
    ($operatingsystem != 'Solaris')
    $kernel in ['linux', 'solaris']
    !str2bool($is_virtual)
```

In the examples above, the operators are `<`, `!=`, `in`, and `!`.

Optionally, expressions can be surrounded by parentheses.

Operands

Operands in an expression may be:

- Literal values
- Variables
- Other expressions
- Function calls which return values

The data type of each operand is dictated by the operator. See the list of operators below for details.

When creating compound expressions by using other expressions as operands, you should use parentheses for clarity:

```
    (90 < 7) and ('Solaris' == 'Solaris') # resolves to false
    (90 < 7) or ('solaris' in ['linux', 'solaris']) # resolves to true
```

Order of Operations

Compound expressions are evaluated in a standard order of operations. However, parentheses will override the order of operations:

```
    # This example will resolve to 30, rather than 23.
    notice( (7+8)*2 )
```

For the sake of clarity, we recommend using parentheses in all but the simplest compound expressions.

The precedence of operators, from highest to lowest:

1. `!` (not)
2. `in`
3. `*` and `/` (multiplication and division)
4. `-` and `+` (addition and subtraction)
5. `<<` and `>>` (left shift and right shift)

6. `==` and `!=` (equal and not equal)
7. `>=`, `<=`, `>`, and `<` (greater or equal, less or equal, greater than, and less than)
8. `and`
9. `or`

Comparison Operators

Comparison operators have the following traits:

- They take operands of several data types
- They resolve to boolean values

`==` (equality)

Resolves to `true` if the operands are equal. Accepts the following types of operands:

- Numbers — Tests simple equality.
- Strings — Case-insensitively tests whether two strings are identical.
- Arrays and hashes — Tests whether two arrays or hashes are identical.
- Booleans — Tests whether two booleans are the same value.

`!=` (non-equality)

Resolves to `false` if the operands are equal. Behaves similarly to `==`.

`<` (less than)

Resolves to `true` if the left operand is smaller than the right operand. Accepts numbers.

The behavior of this operator when used with strings is undefined.□

`>` (greater than)

Resolves to `true` if the left operand is bigger than the right operand. Accepts numbers.

The behavior of this operator when used with strings is undefined.□

`<=` (less than or equal to)

Resolves to `true` if the left operand is smaller than or equal to the right operand. Accepts numbers.

The behavior of this operator when used with strings is undefined.□

`>=` (greater than or equal to)

Resolves to `true` if the left operand is bigger than or equal to the right operand. Accepts numbers.

The behavior of this operator when used with strings is undefined.□

`=~` (regex match)

This operator is non-transitive with regard to data types: it accepts a string as the left operand and a regular expression as the right operand.

Resolves to `true` if the left operand matches the regular expression.

`!~` (regex non-match)

This operator is non-transitive with regard to data types: it accepts a string as the left operand and a regular expression as the right operand.

Resolves to `false` if the left operand matches the regular expression.

`in`

Resolves to `true` if the right operand contains the left operand. This operator is case sensitive.

This operator is non-transitive with regard to data types: it accepts a string as the left operand, and the following types of right operands:

- Strings — Tests whether the left operand is a substring of the right.
- Arrays — Tests whether one of the members of the array is identical to the left operand.
- Hashes — Tests whether the hash has a key named after the left operand.

Examples:

```
    'eat' in 'eaten' # resolves to TRUE
    'Eat' in 'eaten' # resolves to FALSE
    'eat' in ['eat', 'ate', 'eating'] # resolves to TRUE
    'eat' in { 'eat' => 'present tense', 'ate' => 'past tense'} # resolves to
TRUE
    'eat' in { 'present' => 'eat', 'past' => 'ate' } # resolves to FALSE
```

Boolean Operators

Boolean Operators have the following traits:

- They take boolean operands; if another data type is given, it will be automatically converted to boolean
- They resolve to boolean values

These expressions are most useful when creating compound expressions.

`and`

Resolves to `true` if both operands are true, otherwise resolves to `false`.

`or`

Resolves to `true` if either operand is true.

`!` (not)

Takes one operand:

```
$my_value = true
notice ( !$my_value ) # Will resolve to false
```

Resolves to `true` if the operand is false, and `false` if the operand is true.

Arithmetic Operators

Arithmetic Operators have the following traits:

- They take numeric operands
- They resolve to numeric values

`+` (addition)

Resolves to the sum of the two operands.

`-` (subtraction)

Resolves to the difference of the two operands.

`/` (division)

Resolves to the quotient of the two operands.

`*` (multiplication)

Resolves to the product of the two operands.

`<<` (left shift)

Left bitwise shift: shifts the left operand by the number of places specified by the right operand. This is equivalent to rounding each operand down to the nearest integer and multiplying the left operand by 2 to the power of the right operand.

`>>` (right shift)

Right bitwise shift: shifts the left operand by the number of places specified by the right operand. This is equivalent to rounding each operand down to the nearest integer and dividing the left operand by 2 to the power of the right operand.

Backus Naur Form

With the exception of the `in` operator, the available operators in Backus Naur Form are:

```
<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/'
```

Language: Functions

- See the Function Reference for complete info about Puppet's built-in functions.

Functions are pre-defined chunks of Ruby code which run during compilation. Most functions either return values or modify the catalog.

Puppet includes several built-in functions, and more are available in modules on the Puppet Forge, particularly the puppetlabs-stdlib module. You can also write custom functions and put them in your own modules.

Syntax

```
file {'/etc/ntp.conf':
  ensure  => file,
  content => template('ntp/ntp.conf'),
}

include apache2

if str2bool($is_virtual) {
  include ntp::disabled
}
else {
  include ntp
```

```
}
# str2bool is part of the puppetlabs-stdlib module; install it with
# sudo puppet module install puppetlabs-stdlib
```

In the examples above, `template`, `include`, and `str2bool` are all functions. `template` and `str2bool` return values, and `include` modifies the catalog by causing a class to be applied.

The general form of a function call is:

- The name of the function, as a bare word
- An optional opening parenthesis
- Any number of arguments, separated with commas; the number and type of arguments are controlled by the function
- A closing parenthesis, if an open parenthesis was used

Behavior

There are two types of Puppet functions:

- Rvalues return values and can be used anywhere a normal value is expected. (This includes resource attributes, variable assignments, conditions, selector values, the arguments of other functions, etc.) These values can come from a variety of places; the `template` function reads and evaluates a template to return a string, and stdlib's `str2bool` and `num2bool` functions convert values from one data type to another.
- Statements should stand alone and do some form of work, which can be anything from logging a message (like `notice`), to modifying the catalog in progress (like `include`), to causing the entire compilation to fail (`fail`).

All functions run during compilation, which means they can only access the commands and data available on the puppet master. To perform tasks on, or collect data from, an agent node, you must use a resource or a custom fact.

Arguments

Each function defines how many arguments it takes and what data types it expects those arguments to be. These should be documented in the function's `:doc` string, which can be extracted and included in the function reference.

Functions may accept any of Puppet's standard data types. The values passed to the function's Ruby code will be converted to Ruby objects as follows:

Puppet type	Ruby type
boolean	boolean
undef	the empty string

string	string
resource reference	`Puppet::Resource`
number	string
array	array
hash	hash

Language: Classes

Classes are named blocks of Puppet code which are not applied unless they are invoked by name. They can be stored in modules for later use and then declared (added to a node's catalog) with the `include` function or a resource-like syntax.

Syntax

Defining a Class

```
class base::linux {
  file { '/etc/passwd':
    owner => 'root',
    group => 'root',
    mode  => '0644',
  }
  file { '/etc/shadow':
    owner => 'root',
    group => 'root',
    mode  => '0440',
  }
}
```

```
class apache ($version = 'latest') {
  package {'httpd':
    ensure => $version, # Get version from the class declaration
    before => File['/etc/httpd.conf'],
  }
  file {'/etc/httpd.conf':
    ensure  => file,
    owner   => 'httpd',
    content => template('apache/httpd.conf.erb'), # Template from a module
  }
  service {'httpd':
    ensure => running,
    enable => true,
    subscribe => File['/etc/httpd.conf'],
  }
}
```

The general form of a class declaration is:

- The `class` keyword
- The name of the class
- An optional set of parameters, which consists of:
  - An opening parenthesis
  - A comma-separated list of parameters, each of which consists of:
    - A new variable name, including the `$` prefix
    - An optional equals (=) sign and default value (any data type)
  - An optional trailing comma after the last parameter (Puppet 2.7.8 and later)
  - A closing parenthesis
- Optionally, the `inherits` keyword followed by a single class name
- An opening curly brace
- A block of arbitrary Puppet code, which generally contains at least one resource declaration
- A closing curly brace

Declaring a Class With `include`

Declaring a class adds all of the code it contains to the catalog. Classes can be declared with the `include` function function.

```
# Declaring a class with include
include base::linux
```

You can safely use `include` multiple times on the same class and it will only be declared once:

```
include base::linux
include base::linux # Has no additional effect
```

The `include` function can accept a single class name or a comma-separated list of class names:

```
include base::linux, apache # including a list of classes
```

The `include` function cannot pass values to a class's parameters. You may still use `include` with parameterized classes, but only if every parameter has a default value; parameters without defaults are mandatory, and will require you to use the resource-like syntax to declare the class.

Declaring a Class with `require`

The `require` function acts like `include`, but also causes the class to become a dependency of the surrounding container:

```
define apache::vhost ($port, $docroot, $servername, $vhost_name) {
  require apache
  ...
}
```

In the above example, whenever an `apache::vhost` resource is declared, Puppet will add the contents of the `apache` class to the catalog if it hasn't already done so and it will ensure that every resource in class `apache` is processed before every resource in that `apache::vhost` instance.

Note that this can also be accomplished with relationship chaining. The following example will have an identical effect:

```
define apache::vhost ($port, $docroot, $servername, $vhost_name) {
  include apache
  Class['apache'] -> Apache::Vhost[$title]
  ...
}
```

The `require` function should not be confused with the `require` metaparameter.

Declaring a Class Like a Resource

Classes can also be declared like resources, using the special "class" resource type:

```
# Declaring a class with the resource-like syntax
class {'apache':
  version => '2.2.21',
}
# With no parameters:
class {'base::linux':}
```

The parameters used when defining the class become the attributes (without the `$` prefix) available when declaring the class like a resource. Parameters which have a default value are optional; if they are left out of the declaration, the default will be used. Parameters without defaults are mandatory.

A class can only be declared this way once:

```
# WRONG:
class {'base::linux':}
class {'base::linux':} # Will result in a compilation error
```

Thus, unlike with `include`, you must carefully manage where and how classes are declared when

using this syntax.

The resource-like syntax should not be mixed with `include` for a given class. The behavior of the two syntaxes when mixed is undefined; but practically speaking, the results will be parse-order dependent and will sometimes succeed and sometimes fail.

Declaring a Class With an ENC

External node classifiers can declare classes. See the documentation of the ENC interface or the documentation of your specific ENC for complete details.

Note that the ENC API supports classes with or without parameters, but many of the most common ENCs only support classes without parameters.

Behavior

Defining a class makes it available for later use; declaring a class activates it and adds all of its resources to the catalog.

Classes are singletons — although a given class may have very different behavior depending on how it is declared, the resources in it will only be declared once per compilation. You can use `include` several times on the same class, but every time after the first will have no effect. (The `require` function behaves similarly with regards to declaring the class, but will continue to create ordering relationships on subsequent uses.)

Parameters and Attributes

The parameters of a class can be used as local variables inside the class's definition. These variables are not set with normal assignment statements; instead, they are set with attributes when the class is declared:

```
class {'apache':
  version => '2.2.21',
}
```

In the example above, the value of `$version` within the class definition would be set to the attribute `2.2.21`.

Containment

A class contains all of its resources. This means any relationships formed with the class will be extended to every resource in the class.

Note that classes cannot contain other classes. This is a known design issue; see the relevant note on the "Containment" page for more details.

Auto-Tagging

A class's name and each of its namespace segments are automatically added to the tags of every resource it contains.

Metaparameters

When declared with the resource-like syntax, a class may use any metaparameter. In such cases, every resource contained in the class will also have that metaparameter. So if you declare a class with `noop => true`, every resource in the class will also have `noop => true`, unless they specifically override it. Metaparameters which can take more than one value (like the relationship metaparameters) will merge the values from the container and any specific values from the individual resource.

Location

Definitions

Class definitions can (and should) be stored in modules. Puppet is automatically aware of any classes in a valid module and can autoload them by name. Classes should be stored in the `manifests/` directory of a module with one class per file, and each filename should reflect the name of its class; see Module Fundamentals for more details.

> **ASIDE: BEST PRACTICES**
>
> You should usually only load classes from modules. Although the additional options below this aside will work, they are not recommended.

You can also put class definitions in the site manifest. If you do so, they may be placed anywhere in the file and are not parse-order dependent.

This version (2.7) of Puppet still allows class definitions to be stored in other class definitions, which puts the interior class under the exterior class's namespace; it does not cause the interior class to be automatically declared when the exterior class is. Note that although this is not yet formally deprecated, it is very much not recommended.

Declarations

You can declare classes:

- At top scope in the site manifest
- In a node definition
- In the output of an ENC
- In any other class
- In a defined type
- In a conditional statement

If you are using `include` or `require` to declare a class (that is, if you are not declaring it with parameters at any point), you can declare it multiple times in several different places. This is useful for allowing classes or defined types to manage their own dependencies, or for building overlapping "role" classes when a given node may have more than one role. See Aside: History, the Future, and Best Practices below for more information.

Inheritance

Classes can be derived from other classes using the `inherits` keyword. This allows you to make special-case classes that extend the functionality of a more general "base" class.

> Note: Puppet 2.7 does not support using parameterized classes for inheritable base classes. The base class must have no parameters.

Inheritance causes three things to happen:

- When a derived class is declared, its base class is automatically declared first (if it wasn't already declared elsewhere).
- The base class becomes the parent scope of the derived class, so that the new class receives a copy of all of the base class's variables and resource defaults.
- Code in the derived class is given special permission to override any resource attributes that were set in the base class.

> ASIDE: WHEN TO INHERIT
>
> You should only use class inheritance when you need to override resource attributes in the base class. This is because you can instantiate a base class by including it inside another class's definition, and assigning a direct parent scope is rarely necessary since you can use qualified variable names to read any class's internal data.
>
> Additionally, many of the traditional use cases for inheritance (notably the "anti-class" pattern, where you override a service resource's `ensure` attribute to disable it) can be accomplished just as easily with class parameters. It is also possible to use resource collectors to override resource attributes.

Overriding Resource Attributes

The attributes of any resource in the base class can be overridden with a reference to the resource you wish to override, followed by a set of curly braces containing attribute => value pairs:

```
class base::freebsd inherits base::unix {
  File['/etc/passwd'] {
    group => 'wheel'
  }
```

```
    File['/etc/shadow'] {
      group => 'wheel'
    }
  }
```

This is identical to the syntax for adding attributes to an existing resource, but in a derived class, it gains the ability to rewrite resources instead of just adding to them. Note that you can also use multi-resource references here.

You can remove an attribute's previous value without setting a new one by overriding it with the special value `undef`:

```
  class base::freebsd inherits base::unix {
    File['/etc/passwd'] {
      group => undef,
    }
  }
```

This causes the attribute to be unmanaged by Puppet.

Appending to Resource Attributes

Some resource attributes (such as the relationship metaparameters) can accept multiple values in an array. When overriding attributes in a derived class, you can add to the existing values instead of replacing them by using the `+>` ("plusignment") keyword instead of the standard `=>` hash rocket:

```
  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'], File['httpd.conf'] ],
      # Since `require` will retain its previous values, this is equivalent
to:
      # require => [ Package['httpd'], File['apache.pem'], File['httpd.conf']
],
    }
  }
```

Aside: History, the Future, and Best Practices

Classes often need to be configured with site-specific and node-specific data, especially if they are to be re-used at multiple sites.

The traditional way to get this info into a class was to have it look outside its local scope and read arbitrary variables, which would be set by the user however they saw fit. (If you're curious, this was why ENC-set variables were originally called "parameters:" they were almost always used to pass data into classes.) This entire approach was brittle and bad, because all classes were effectively competing for variable names in a global namespace, and the only ways to find a given class's requirements were to be really diligent about documentation or read the entire module's code.

Parameters for classes were introduced in Puppet 2.6 as a way to directly pass site/node-specific data into a class. By declaring up-front what information was necessary to configure the class, module developers could communicate quickly and unambiguously to users and Puppet Labs could eventually build automated tooling to help with discovery. This helped a bit, but it also introduced new problems and revealed some existing ones:

- Given that the `include` function can take multiple classes, there was no good way to make it also accept class parameters. This necessitated a new and less convenient syntax for using classes with parameters.
- If a class were to be declared twice with conflicting parameter values, there was no framework for deciding which declaration should win. Thus, Puppet will simply fail compilation if there's any possibility of a conflict — that is, if the syntax that allows parameters is used twice for a given class. The result: parameterized classes wouldn't work with some very common design patterns, including:
  - Having classes and defined types `include` or `require` any classes they depend on.
  - Building overlapping "role" classes and declaring more than one role on some nodes.
- The question of what to do about parameter conflicts also emphasized the fact that, using the traditional method of grabbing arbitrary variables, it was already possible to create parse-order dependent conflicts by using `include` multiple times in different scopes. (This remained possible after parameterized classes were introduced.)

The result was that class parameters were an incomplete feature, which didn't finish solving the problems that inspired them — or rather, they could solve the problem, but the cost was a much more difficult and rigorous site design, one which felt unnatural to many users. This actually made the problem a bit worse, mostly by muddying our message to users about how to deal with these issues and presenting the illusion that a very-much-still-alive problem was solved. This remained the state of affairs in Puppet 2.7.

After a lot of research, we decided there were actually two requirements for really solving the question of site/node-specific class data. The first was explicit class parameters, which we now had; the second was a guarantee that, while compiling a given node's catalog, there would only be one possible value for any given parameter. This second piece of the puzzle would restore and reaffirm the usefulness of `include`, let parameterized classes work with the traditional large-scale Puppet design patterns, and still let us have all of the benefits of

class parameters. (That is: strict namespacing, obvious placement, and visibility to outside tools.) Since Puppet's language allows so much flexible logic in manifests, we determined that the only way to fulfill this second requirement was to fetch parameter values from somewhere outside the Puppet manifests, and the fits-most-cases tool we settled on was Hiera.

Puppet 3.0 will get closer to solving this question with automatic parameter lookup, which will work as follows:

- Puppet will require Hiera, a hierarchical data lookup tool which lets you set site-wide values and override them for groups of nodes and specific nodes.
- The `include` function can be used with every class, including parameterized classes.
- If you use `include` on a class with parameters, Puppet will automatically look up each parameter in Hiera, using the lookup key `class_name::parameter_name`. (So the `apache` class's `$version` parameter would be looked up as `apache::version`.) If a parameter isn't set in Hiera, Puppet will use the default value; if it's absent and there's no default, compilation will fail and you'll have to set a value for it if you want to use the class.
- You can still set parameters directly with the resource-like syntax or with an ENC and they will override any values from Hiera. However, you shouldn't need to (and won't want to) do this.

In the meantime, there are several approaches to dealing with this space in your own modules and site design.

Best Practices Today

In a Puppet 2.7 or 2.6 world, you have the following general options:

- Use Hiera and parameterized classes to mimic 3.0 behavior in a forward-compatible way
- Use a "classic" module design that doesn't use parameterized classes
- Use a rigorous "pure" parameterized classes site design, probably using an ENC to resolve parameters and machine roles
- Mix and match "classic" and parameterized classes, using parameters only where necessary and Hiera when you feel like it

USING HIERA TO MIMIC 3.0

Hiera works today as an add-on with Puppet 2.7 and 2.6. If you maintain site data in Hiera and write your parameterized classes to use the following idiom, you can have a complete forward-compatible emulation of Puppet 3.0's auto-lookup:

```
    class example ( $parameter_one = hiera('example::parameter_one'),
$parameter_two = hiera('example::parameter_two') ) {
      ...
    }
```

This allows you to use `include` on the class and automatically retrive parameter values from Hiera. When you upgrade to 3.0, Puppet will begin automatically looking up the exact same values that it was manually looking up in Puppet 2.7; you can remove the `hiera()` statements in your default values at your leisure, or leave them there for the sake of of backwards compatibility.

See the Hiera documentation for more details about storing your data in Hiera.

USING "CLASSIC" MODULE/SITE DESIGN

Continue to use classes with no parameters, and have them fetch their data from variables outside their scope. This will have the same drawbacks it has always had, and you will need to beware the temptation to abuse dynamic scope. We highly recommend fetching these node- or group-specific values from an ENC instead of calculating them with scope hierarchies in Puppet. You may also find the classic `extlookup` function (or Hiera as an add-on) helpful, and many users have built Puppet function interfaces to an external CMDB data source for this exact purpose.

USING A "PURE" PARAMETERIZED CLASSES SITE

In short, you'll need to do the following:

- Abandon `include` and use the resource-like syntax to declare all classes.
- If you use "role" classes, make them granular enough that they have absolutely no overlap. Each role class should completely "own" the parameterized classes it declares and nodes (via node definitions or your ENC) can declare whichever roles they need.
- If you don't use "role" classes, every node should declare every single class it needs. This is extraordinarily unwieldy with node definitions, and you will almost certainly need a custom-built ENC which can resolve classes and parameters in a hierarchical fashion.
- Most of your non-role classes or defined types shouldn't declare other classes. If any of them require a given class, you should establish a dependency relationship with the chaining syntax inside the definition (`Class['dependency'] -> Class['example']` or `Class['dependency'] -> Example::Type[$title]`) — this won't declare the class in question, but will fail compilation if the class isn't being declared elsewhere (such as in the role, node definition, or ENC).
- If a class does declare another class, it must "own" that class completely, in the style of the "`ntp`, `ntp::service`, `ntp::config`, `ntp::package`" design pattern.

Most users will want to do something other than this, as it takes fairly extreme design discipline. However, once constructed, it is reliable and forward-compatible.

MIXING AND MATCHING

The most important thing when mixing styles is to make sure your site's internal documentation is very, very clear.

If possible, you should implement Hiera and use the idiom above for mimicking 3.0 behavior

on your handful of parameterized classes. This will give you and your colleagues an obvious path forward when you eventually refactor your existing modules, and you can safely continue to add parameters (or not) at your leisure while declaring all of your classes in the same familiar way.

Language: Defined Resource Types

Defined resource types (also called defined types or defines) are blocks of Puppet code that can be evaluated multiple times with different parameters. Once defined, they act like a new resource type: you can cause the block to be evaluated by declaring a resource of that new type.

Defines can be used as simple macros or as a lightweight way to develop fairly sophisticated resource types.

Syntax

Defining a Type

```
    # /etc/puppetlabs/puppet/modules/apache/manifests/vhost.pp
    define apache::vhost ($port, $docroot, $servername = $title, $vhost_name =
'*') {
      include apache # contains Package['httpd'] and Service['httpd']
      include apache::params # contains common config settings
      $vhost_dir = $apache::params::vhost_dir
      file { "${vhost_dir}/${servername}.conf":
        content => template('apache/vhost-default.conf.erb'),
          # This template can access all of the parameters and variables from
above.
        owner   => 'www',
        group   => 'www',
        mode    => '644',
        require => Package['httpd'],
        notify  => Service['httpd'],
      }
    }
```

This creates a new type called `apache::vhost`.

The general form of a type definition is:

- The `define` keyword
- The name of the defined type
- An optional set of parameters, which consists of:
  - An opening parenthesis
  - A comma-separated list of parameters, each of which consists of:

    - A new variable name, including the `$` prefix
    - An optional equals sign and default value (any data type)
  - An optional trailing comma after the last parameter (Puppet 2.7.8 and later)
  - A closing parenthesis
- An opening curly brace
- A block of arbitrary Puppet code, which generally contains at least one resource declaration
- A closing curly brace

The definition does not cause the code in the block to be added to the catalog; it only makes it available. To execute the code, you must declare one or more resources of the defined type.

Declaring an Instance

Instances of a defined type (often just called "resources") can be declared the same way a normal resource is declared. (That is, with a type, title, and set of attribute/value pairs.)

The parameters used when defining the type become the attributes (without the `$` prefix) used when declaring resources of that type. Parameters which have a default value are optional; if they are left out of the declaration, the default will be used. Parameters without defaults must be specified.

To declare a resource of the `apache::vhost` type from the example above:

```
apache::vhost {'homepages':
  port    => 8081,
  docroot => '/var/www-testhost',
}
```

Behavior

If a defined type is present, you can declare resources of that type anywhere in your manifests. See "Location" below for details.

Declaring a resource of the type will cause Puppet to re-evaluate the block of code in the definition, using different values for the parameters.

Parameters and Attributes

Every parameter of a defined type can be used as a local variable inside the definition. These variables are not set with normal assignment statements; instead, each instance of the defined type uses its attributes to set them:

```
apache::vhost {'homepages':
  port    => 8081, # Becomes the value of $port
```

```
        docroot => '/var/www-testhost', # Becomes the value of $docroot
      }
```

`$title` and `$name`

Every defined type gets two “free” parameters, which are always available and do not have to be explicitly added to the definition:

- `$title` is always set to the title of the instance. Since it is guaranteed to be unique for each instance, it is useful when making sure that contained resources are unique. (See “Resource Uniqueness” below.)
- `$name` defaults to the value of `$title`, but users can optionally specify a different value when they declare an instance. This is only useful for mimicking the behavior of a resource with a namevar, which is usually unnecessary. If you are wondering whether to use `$name` or `$title`, use `$title`.

Unlike the other parameters, the values of `$title` and `$name` are already available inside the parameter list. This means you can use `$title` as the default value (or part of the default value) for another attribute:

```
    define apache::vhost ($port, $docroot, $servername = $title, $vhost_name =
'*') { ...
```

Resource Uniqueness

Since multiple instances of a defined type might be declared in your manifests, you must make sure that every resource in the definition will be different in every instance. Failing to do this will result in compilation failures with a “duplicate resource declaration” error.

You can make resources different across instances by making their titles and names/namevars include the value of `$title` or another parameter.

```
    file { "${vhost_dir}/${servername}.conf":
```

Since `$title` (and possibly other parameters) will be unique per instance, this ensures the resources will be unique as well.

Containment

Every instance of a defined type contains all of its unique resources. This means any relationships formed between the instance and another resource will be extended to every resource that makes up the instance.

Metaparameters

The declaration of a defined type instance can include any metaparameter. If it does:

- Every resource contained in the instance will also have that metaparameter. So if you declare a defined resource with `noop => true`, every resource contained in it will also have `noop => true`, unless they specifically override it. Metaparameters which can take more than one value (like the relationship metaparameters) will merge the values from the container and any specific values from the individual resource.
- The value of the metaparameter can be used as a variable in the definition, as though it were a normal parameter. (For example, in an instance declared with `require => Class['ntp']`, the local value of `$require` would be `Class['ntp']`.)

Resource Defaults

Just like with a normal resource type, you can declare resource defaults for a defined type:

```
# /etc/puppetlabs/puppet/manifests/site.pp
Apache::Vhost {
  port => 80,
}
```

In this example, every resource of the type would default to port 80 unless specifically overridden.

Location

Defined types can (and should) be stored in modules. Puppet is automatically aware of any defined types in a valid module and can autoload them by name. Definitions should be stored in the `manifests/` directory of a module with one definition per file and each filename should reflect the name of its type. See Module Fundamentals for more details.

> **ASIDE: BEST PRACTICES**
>
> You should usually only load defined types from modules. Although the additional options below this aside will work, they are not recommended.

You can also put type definitions in the site manifest. If you do so, they may be placed anywhere in the file and are not parse-order dependent.

Type definitions may also be placed inside class definitions; however, this limits their availability to that class and is not recommended for any purpose. This is not formally deprecated in this version of Puppet (2.7), but may become so in a future release.

Naming

The characters allowed in a defined type's name are listed here.

If the definition is stored in a module, its name must reflect its place in the module with its

namespace. See Module Fundamentals for details.

Note that if a type's name has one or more namespaces in it, each name segment must be capitalized when writing a resource reference, collector, or resource default. (For example, a reference to the vhost resource declared above would be `Apache::Vhost['homepages']`.)

Language: Containment of Resources

Containment

Classes and defined type instances contain the resources they declare. This means that if any resource or class forms a relationship with the container, it will form the same relationship with every resource inside the container.

```
class ntp {
  file {'/etc/ntp.conf':
    ...
    require => Package['ntp'],
    notify  => Service['ntp'],
  }
  service {'ntp':
    ...
  }
  package {'ntp':
    ...
  }
}

include ntp
exec {'/usr/local/bin/update_custom_timestamps.sh':
  require => Class['ntp'],
}
```

In this example, `Exec['/usr/local/bin/update_custom_timestamps.sh']` would happen after every resource in the ntp class, including the package, the file, and the service.

This feature also allows you to notify and subscribe to classes and defined resource types as though they were a single resource.

Known Issues

Classes do not get contained by the class or defined type that declares them. This is a known design problem, and can be tracked at issue #8040.

```
class ntp {
  include ntp::conf_file
```

```
    service {'ntp':
      ...
    }
    package {'ntp':
      ...
    }
  }
```

In the above example, a resource with a `require => Class['ntp']` metaparameter would be applied after both `Package['ntp']` and `Service['ntp']`, but would not necessarily happen after any of the resources contained by the `ntp::conf_file` class; those resources would "float off" outside the NTP class.

Context and Plans

Containment is a singleton and is absolute: a resource can only be contained by one container (although the container, in turn, may be contained). However, classes can be declared in multiple places with the `include` function. A naïve interpretation would thus imply that classes can be in multiple containers at once.

Puppet 0.25 and prior would establish a containment edge with the first container in which a class was declared. This made containment dependent on parse-order, which was bad. However, fixing this unpredictability in 2.6 left no native way for the main "public" class in a module to completely own its subordinate implementation classes. This makes it hard to keep very large modules readable, since it complicates and obscures logical relationships in large blocks of code.

Puppet Labs is investigating ways to resolve this for a future Puppet version.

Workaround: The Anchor Pattern

You can cause a class to act like it's contained in another class by "holding it in place" with both a `require` and `before` relationship to resources that ARE contained:

```
  class ntp {
    include ntp::conf_file

    package {'ntp':
      ...
      before => Class['ntp::conf_file'],
    }
    service {'ntp':
      ...
      subscribe => Class['ntp::conf_file'],
      # Remember that 'subscribe' is effectively 'require, and also...'
    }
  }
```

In this case, the `ntp::conf_file` class still isn't technically contained, but any resource can safely form a relationship with the `ntp` class and rest assured that the relationship will propagate into all relevant resources.

Since this anchoring behavior is effectively an invisible side effect of the relationships inside the class, you should not rely on relationships with normal resources. Instead, you should use the `anchor` resource type included in the puppetlabs-stdlib module, which exists solely for this purpose.

Language: Namespaces and Autoloading

Class and defined type names may be broken up into segments called namespaces. Namespaces tell the autoloader how to find the class or defined type in your modules.

> Important note: Earlier versions of Puppet used namespaces to navigate nested class/type definitions, and the code that resolves names still behaves as though this were their primary use. This can sometimes result in the wrong class being loaded. This is a major outstanding design issue (issue #2053) which will not be resolved in Puppet 2.7. See below for a full description of the issue.

Syntax

Puppet class and defined type names may consist of any number of namespace segments separated by the `::` (double colon) namespace separator. (This separator is analogous to the `/` [slash] in a file path.)

```
class apache { ... }
class apache::mod { ... }
class apache::mod::passenger { ... }
define apache::vhost { ... }
```

Optionally, class/define names can begin with the top namespace, which is the empty string. The following names are equivalent:

- `apache` and `::apache`
- `apache::mod` and `::apache::mod`
- etc.

This is ugly and should be unnecessary, but is occasionally required due to an outstanding design issue. See below for details.

Autoloader Behavior

When a class or defined resource is declared, Puppet will use its full name to find the class or defined type in your modules. Names are interpreted as follows:

- The first segment in a name (excluding the empty "top" namespace) identifies the module. Every class and defined type should be in its own file in the module's `manifests` directory, and each file should have the `.pp` file extension.
- If there are no additional namespaces, Puppet will look for the class or defined type in the module's `init.pp` file.
- Otherwise, Puppet will treat the final segment as the file name and any interior segments as a series of subdirectories under the `manifests` directory.

Thus, every class or defined type name maps directly to a file path within Puppet's `modulepath`:

name	file path
`apache`	`<modulepath>/apache/manifests/init.pp`
`apache::mod`	`<modulepath>/apache/manifests/mod.pp`
`apache::mod::passenger`	`<modulepath>/apache/manifests/mod/passenger.pp`

Note again that `init.pp` always contains a class or defined type named after the module, and any other `.pp` file contains a class or type with at least two namespace segments. (That is, `apache.pp` would contain a class named `apache::apache`.)

Relative Name Lookup and Incorrect Name Resolution

In Puppet 2.7, class name resolution is partially broken — if the final namespace segment of a class in one module matches the name of another module, Puppet will sometimes load the wrong class.

```
class bar {
  notice("From class bar")
}
class foo::bar {
  notice("From class foo::bar")
}
class foo {
  include bar
}
include foo
```

In the example above, the invocation of `include bar` will actually declare class `foo::bar`. This is because Puppet assumes class and defined type names are relative until proven otherwise. This is a major outstanding design issue (issue #2053) which will not be resolved in Puppet 2.7, as the fix will break a large amount of existing code and require a long deprecation period.

Behavior

When asked to load a class or defined type `foo`, Puppet will:

- Attempt to load `<current namespace>::foo`
- If that fails, attempt to load `<parent of current namespace>::foo`
- If that fails, continue searching for `foo` through every ancestor namespace
- Finally, attempt to load `foo` from the top namespace (AKA `::foo`)

A concrete example:

```
class apache::nagios {
  include nagios
  ...
}
```

When asked to `include nagios`, Puppet will first attempt to load `apache::nagios::nagios`. Since that class does not exist, it will then attempt to load `apache::nagios`. This exists, and since the include function can safely declare a class multiple times, Puppet does not complain. It will not attempt to load class `nagios` from the `nagios` module.

Workaround

If a class within another module is blocking the declaration of a top-namespace class, you can force the correct class to load by specifying its name from the top namespace (as seen above). To specify a name from the top namespace, prepend `::` (double colon) to it:

```
class apache::nagios {
  include ::nagios # Start searching from the top namespace instead of the
local namespace
  ...
}
```

In the example above, Puppet will load class `nagios` from the `nagios` module instead of declaring `apache::nagios` a second time.

Aside: Historical Context

Relative name lookup was introduced in pre-module versions of Puppet. It reflects an outdated assumption about how modules would be used.

PROTO-MODULES

Before modules were introduced, users would create module-like blobs by putting a group

of related classes and defined types into one manifest file, then using an import statement in `site.pp` to make the group available to the parser.

```
# /etc/puppet/manifests/apache.pp
class apache { ... } # Manage Apache
class ssl { ... } # Optional SSL support for Apache
class python { ... } # Optional mod_python support for Apache
define vhost ($port) { ... } # Create an Apache vhost

# /etc/puppet/manifests/site.pp
import apache.pp
include apache
include ssl
```

NAMESPACING FOR REDISTRIBUTION

As proto-modules got more sophisticated, their authors wanted to share them with other users. The problem with this is visible above: many modules were likely to have a `python` or `ssl` class, and the `lighttpd` module probably had a `vhost` define that clashed with the Apache one.

The solution was namespacing, which would allow different proto-modules to use common class and defined type names without competing for global identifiers.

PRIVATE VS. PUBLIC

The implementation of namespaces relied on an assumption that turned out to be incorrect: that classes and defined types other than the module's main class would (and should) mostly be used inside the module, rather than applied directly to nodes. (That is, they would be private, much like local variables.) Thus, namespacing was done by hiding definitions within other definitions:

```
class apache {
  ...
  class ssl { ... }
  class python { ... }
  define vhost ($port) { ... }
}
```

The short names of the internal classes and defined types could only be used inside the main class. However, much like qualified variables, you could access them from anywhere by using their full (that is, namespaced) name. Full names were constructed by prepending the full name of the "outer" class, along with the `::` namespace separator. (That is, the full name of `ssl` would be `apache::ssl`, `python` would be `apache::python`, etc.)

This was the origin of the relative name lookup behavior, as Puppet assumed that a class that had its own private `python` class would want to use that instead of the top-namespace `python` class.

THIS TURNED OUT TO BE POINTLESS

Users and developers eventually realized several things about this arrangement:

- Using a class's full name everywhere was actually not that big a deal and was in fact a lot clearer and easier to read and maintain.
- Public classes and defined types were more common than private ones and optimizing for the less common case was an odd approach.
- Even for classes and defined types that were only used within their module, there was little real benefit to be gained by making them "private," since they were effectively public via their full name anyway.

Those realizations led to the superior module autoloader design used today, where a class's "full" name is effectively its only name. However, the previous name lookup behavior was never deprecated or removed, for fear of breaking large amounts of existing code. This leaves it present in Puppet 2.7, where it often annoys users who have adopted the modern code style.

We plan to fix this in a future release, after a suitable deprecation period.

Language: Resource Collectors

Resource collectors (AKA the spaceship operator) select a group of resources by searching the attributes of every resource in the catalog. This search is parse-order independent (that is, it even includes resources which haven't yet been declared at the time the collector is written). Collectors realize virtual resources, can be used in chaining statements, and can override resource attributes.

Collectors have an irregular syntax that lets them function as both a statement and a value.

Syntax

```
    User <| title == 'luke' |> # Will collect a single user resource whose
title is 'luke'
    User <| groups == 'admin' |> # Will collect any user resource whose list of
supplemental groups includes 'admin'
    Yumrepo['custom_packages'] -> Package <| tag == 'custom' |> # Will create
an order relationship with several package resources
```

The general form of a resource collector is:

- The resource type, capitalized

- `<|` — An opening angle bracket (less-than sign) and pipe character
- Optionally, a search expression (see below)
- `|>` — A pipe character and closing angle bracket (greater-than sign)

Note that exported resource collectors have a slightly different syntax; see below.

Search Expressions

Collectors can search the values of resource titles and attributes using a special expression syntax. This resembles the normal syntax for Puppet expressions, but is not the same.

> Note: Collectors can only search on attributes which are present in the manifests and cannot read the state of the target system. For example, the collector `Package <| provider == yum |>` would only collect packages whose `provider` attribute had been explicitly set to `yum` in the manifests. It would not match any packages that would default to the `yum` provider based on the state of the target system.

A collector with an empty search expression will match every resource of the specified type.

Parentheses may be used to improve readability. You can create arbitrarily complex expressions using the following four operators:

- `==`
- `!=`
- `and`
- `or`

`==` (EQUALITY SEARCH)

This operator is non-transitive:

- The left operand (attribute) must be the name of a resource attribute or the word `title` (which searches on the resource's title).
- The right operand (search key) must be a string, boolean, number, resource reference, or undef. The behavior of arrays and hashes in the right operand is undefined in this version (2.7) of Puppet.

For a given resource, this operator will match if the value of the attribute (or one of the value's members, if the value is an array) is identical to the search key.

`!=` (NON-EQUALITY SEARCH)

This operator is non-transitive:

- The left operand (attribute) must be the name of a resource attribute or the word `title` (which

searches on the resource's title).

- The right operand (search key) must be a string, boolean, number, resource reference, or undef. The behavior of arrays and hashes in the right operand is undefined in this version (2.7) of Puppet.

For a given resource, this operator will match if the value of the attribute is not identical to the search key.

> Note: This operator will always match if the attribute's value is an array. This behavior may be undefined.

`AND`

Both operands must be valid search expressions.

For a given resource, this operator will match if both of the operands would match for that resource.

`OR`

Both operands must be valid search expressions.

For a given resource, this operator will match if either of the operands would match for that resource.

Location

Resource collectors may be used as independent statements, as the operand of a chaining statement, or in a collector attribute block for amending resource attributes.

Notably, collectors cannot be used as the value of a resource attribute, the argument of a function, or the operand of an expression.

Behavior

A resource collector will always realize any virtual resources that match its search expression. Note that empty search expressions match every resource of the specified type.

In addition to realizing, collectors can function as a value in two places:

- When used in a chaining statement, a collector will act as a proxy for every resource (virtual or non) that matches its search expression.
- When given a block of attributes and values, a collector will set and override those attributes for every resource (virtual or not) that matches its search expression.

Note again that collectors used as values will also realize any matching virtual resources. If you use virtualized resources, you must use care when chaining collectors or using them for overrides.

Exported Resource Collectors

An exported resource collector uses a modified syntax that realizes exported resources.

Syntax

Exported resource collectors are identical to collectors, except that their angle brackets are doubled.

```
Nagios_service <<| |>> # realize all exported nagios_service resources
```

The general form of an exported resource collector is:

- The resource type, capitalized
- `<<|` — Two opening angle brackets (less-than signs) and a pipe character
- Optionally, a search expression (see above)
- `|>>` — A pipe character and two closing angle brackets (greater-than signs)

Behavior

Exported resource collectors exist only to import resources that were published by other nodes. To use them, you need to have resource stashing (storeconfigs) enabled. See Exported Resources for more details. To enable resource stashing, follow the installation instructions and Puppet configuration instructions in the PuppetDB manual.

Like normal collectors, exported resource collectors can be used with attribute blocks and chaining statements.

Language: Node Definitions

A node definition or node statement is a block of Puppet code that will only be included in one node's catalog. This feature allows you to assign specific configurations to specific nodes.

Node statements are an optional feature of Puppet. They can be replaced by or combined with an external node classifier, or you can eschew both and use conditional statements with facts to classify nodes.

Unlike more general conditional structures, node statements only match nodes by name. By default, the name of a node is its certname (which defaults to the node's fully qualified domain name).

Location

Node definitions should go in the site manifest (site.pp).

Alternately, you can store node definitions in any number of manifest files which are imported into

site.pp:

```
# /etc/puppetlabs/puppet/manifests/site.pp

# Import every file in /etc/puppetlabs/puppet/manifests/nodes/
# (Usually, each file contains one node definition.)
import 'nodes/*.pp'

# Import several nodes from a single file
import 'extra_nodes.pp'
```

This is one of the only recommended use cases for `import`. Note that using `import` will require you to restart the puppet master if you change the node manifests and that importing many files will slow down Puppet's compilation time. See the documentation of `import` for details.

> Node statements should never be put in modules. The behavior of a node statement in an autoloaded manifest is undefined.

Syntax

```
# /etc/puppetlabs/puppet/manifests/site.pp
node 'www1.example.com' {
  include common
  include apache
  include squid
}
node 'db1.example.com' {
  include common
  include mysql
}
```

In the example above, only `www1.example.com` would receive the apache and squid classes, and only `db1.example.com` would receive the mysql class.

Node definitions look like class definitions. The general form of a node definition is:

- The `node` keyword
- The name(s) of the node(s)
- Optionally, the `inherits` keyword followed by the name of another node definition
- An opening curly brace
- Any mixture of class declarations, variables, resource declarations, collectors, conditional statements, chaining relationships, and functions
- A closing curly brace

ASIDE: BEST PRACTICES

Although node statements can contain almost any Puppet code, we recommend that you only use them to set variables and declare classes. Avoid using resource declarations, collectors, conditional statements, chaining relationships, and functions in them; all of these belong in classes or defined types.

This will make it easier to switch between node definitions and an ENC.

Naming

Node statements match nodes by name. A node's name is its unique identifier; by default, this is its certname setting, which in turn resolves to the node's fully qualified domain name.

NOTES ON NODE NAMES

- The set of characters allowed in a node name is undefined in this version of Puppet. For best future compatibility, you should limit node names to letters, numbers, periods, underscores, and dashes.
- Although it is possible to configure Puppet to use something other than the certname as a node name, this is not generally recommended.

A node statement's name must be one of the following:

- A quoted string
- The bare word `default`
- A regular expression

You may not create two node statements with the same name.

Multiple Names

You can use a comma-separated list of names to create a group of nodes with a single node statement:

```
node 'www1.example.com', 'www2.example.com', 'www3.example.com' {
  include common
  include apache, squid
}
```

This example creates three identical nodes: `www1.example.com`, `www2.example.com`, and `www3.example.com`.

The Default Node

The name `default` (without quotes) is a special value for node names. If no node statement matching a given node can be found, the `default` node will be used. See Behavior below.

Regular Expression Names

Regular expressions (regexes) can be used as node names. This is another method for writing a single node statement that matches multiple nodes.

```
node /^www\d+$/ {
  include common
}
```

The above example would match `www1`, `www13`, and any other node whose name consisted of `www` and one or more digits.

```
node /^(foo|bar)\.example\.com$/ {
  include common
}
```

The above example would match `foo.example.com` and `bar.example.com`, but no other nodes.

> Make sure that node regexes do not overlap. If more than one regex statement matches a given node, the one it gets will be parse-order dependent.

NO REGEX CAPTURE VARIABLES

Regular expression node names do not use numbered variables to expose captures from the pattern inside the node definition. This differs from the behavior of conditional statements that use regexes.

Behavior

If site.pp contains at least one node definition, it must have one for every node; compilation for a node will fail if one cannot be found. (Hence the usefulness of the `default` node.) If site.pp contains no node definitions, this requirement is dropped.

Matching

A given node will only get the contents of one node definition, even if two node statements could match a node's name. Puppet will do the following checks in order when deciding which definition to use:

1. If there is a node definition with the node's exact name, Puppet will use it.
2. If there is at least one regular expression node statement that matches the node's whole name, Puppet will use the first one it finds.

3. If the node's name looks like a fully qualified domain name (i.e. multiple period-separated groups of letters, numbers, underscores and dashes), Puppet will chop off the final group and start again at step 1. (That is, if a definition for `www01.example.com` isn't found, Puppet will look for a definition matching `www01.example`.)
4. Puppet will use the `default` node.

Thus, for the node `www01.example.com`, Puppet would try the following, in order:

- `www01.example.com`
- The first regex matching `www01.example.com`
- `www01.example`
- The first regex matching `www01.example`
- `www01`
- The first regex matching `www01`
- `default`

You can turn off this fuzzy name matching by changing the puppet master's `strict_hostname_checking` setting to `true`. This will cause Puppet to skip step 3 and only use the node's full name before resorting to `default`.

Code Outside Node Statements

Puppet code that is outside any node statement will be compiled for every node. That is, a given node will get both the code in its node definition and the code outside any node definition.

Node Scope

Node definitions create a new anonymous scope that can override variables and defaults from top scope. See the section on node scope for details.

Merging With ENC Data

Node definitions and external node classifiers can co-exist. Puppet merges their data as follows:

- Variables from an ENC are set at top scope and can thus be overridden by variables in a node definition.
- Classes from an ENC are declared at node scope, which means they will be affected by any variables set in the node definition.

Although ENCs and node definitions can work together, we recommend that most users pick one or the other.

Inheritance

Nodes can inherit from other nodes using the `inherits` keyword. Inheritance works identically to class inheritance. This feature is not recommended; see the aside below.

Example:

```
node 'common' {
  $ntpserver = 'time.example.com'
  include common
}
node 'www1.example.com' inherits 'common' {
  include ntp
  include apache
  include squid
}
```

In the above example, `www1.example.com` would receive the `common, ntp, apache,` and `squid` classes, and would have an `$ntpserver` of `time.example.com`.

ASIDE: BEST PRACTICES

You should almost certainly avoid using node inheritance. Many users attempt to do the following:

```
node 'common' {
  $ntpserver = 'time.example.com'
  include common
  include ntp
}
node 'www01.example.com' inherits 'common' {
  # Override default NTP server:
  $ntpserver = '0.pool.ntp.org'
}
```

This will have the opposite of the intended effect, because Puppet treats node definitions like classes. It does not mash the two together and then compile the mix; instead, it compiles the base class, then compiles the derived class, which gets a parent scope and special permission to modify resource attributes from the base class.

In the example above, this means that by the time `node www01.example.com` has set its own value for `$ntpserver`, the `ntp` class has already received the value it needed and is no longer interested in that variable. For the derived node to override that variable for classes in the base node, it would have to be complied before the base node, and there is no way for Puppet's current implementation to do that.

ALTERNATIVES TO NODE INHERITANCE

- Most users who need hierarchical data should keep it in an external source and have their manifests look it up. The best solution right now is Hiera, which is available as an add-on for Puppet 2.7 and will be available by default in Puppet 3.0. You can also use the

extlookup function, which is available by default in Puppet 2.6 and later.

- ENCs can look up data from any arbitrary source, and return it to Puppet as top-scope variables.
- If you have node-specific data in an external CMDB, you can easily write custom Puppet functions to query it.
- For very small numbers of nodes, you can copy and paste to make complete node definitions for special-case nodes.
- With discipline, you can use node inheritance only for data lookup. The safest approach is to only set variables in the base nodes, then declare all classes in the derived nodes. This is less terse than the mix-and-match that most users try first, but is completely reliable.

Language: Data Types

The Puppet language allows several data types as variables, attribute values, and function arguments:

Booleans

The boolean type has two possible values: `true` and `false`. Literal booleans must be one of these two bare words (that is, not quoted).

The condition of an "if" statement is a boolean value. All of Puppet's comparison expressions return boolean values, as do many functions.

Automatic Conversion to Boolean

If a non-boolean value is used where a boolean is required, it will be automatically converted to a boolean as follows:

Strings

Empty strings are false; all other strings are true. That means the string `"false"` actually resolves as true. Warning: all facts are strings in this version of Puppet, so "boolean" facts must be handled carefully.

> Note: the puppetlabs-stdlib module includes a `str2bool` function which converts strings to boolean values more intelligently.

Numbers

All numbers are true, including zero and negative numbers.

> Note: the puppetlabs-stdlib module includes a `num2bool` function which converts

> > numbers to boolean values more intelligently.

Undef

> The special data type `undef` is false.

Arrays and Hashes

> Any array or hash is true, including the empty array and empty hash.

Resource References

> Any resource reference is true, regardless of whether or not the resource it refers to has been evaluated, whether the resource exists, or whether the type is valid.

Regular expressions cannot be converted to boolean values.

Undef

Puppet's special undef value is roughly equivalent to nil in Ruby; variables which have never been declared have a value of `undef`. Literal undef values must be the bare word `undef`.

The undef value is usually useful for testing whether a variable has been set. It can also be used as the value of a resource attribute, which can let you un-set any value inherited from a resource default and cause the attribute to be unmanaged.

When used as a boolean, `undef` is false.

Strings

Strings are unstructured text fragments of any length. They may or may not be surrounded by quotation marks. Use single quotes for all strings that do not require variable interpolation, and double quotes for strings that do require variable interpolation.

Bare Words

Bare (that is, not quoted) words are usually treated as single-word strings. To be treated as a string, a bare word must:

- Not be a reserved word
- Begin with a lower case letter, and contain only letters, digits, hyphens (-), and underscores (_). Bare words that begin with upper case letters are interpreted as resource references.

Bare word strings are usually used with attributes that accept a limited number of one-word values, such as `ensure`.

Single-Quoted Strings

Strings surrounded by single quotes `'like this'` do not interpolate variables, and the only escape sequences permitted are `\'` (a literal single quote) and `\\` (a literal backslash). Line breaks within the string are interpreted as literal line breaks.

Lone backslashes are literal backslashes, unless followed by a single quote or another backslash. That is:

- When a backslash occurs at the very end of a single-quoted string, a double backslash must be used instead of a single backslash. For example: `path => 'C:\Program Files(x86)\\'`
- When a literal double backslash is intended, a quadruple backslash must be used.

Double-Quoted Strings

Strings surrounded by double quotes `"like this"` allow variable interpolation and several escape sequences. Line breaks within the string are interpreted as literal line breaks, and you can also insert line breaks by using the `\n` escape sequence.

VARIABLE INTERPOLATION

Any `$variable` in a double-quoted string will be replaced with its value. To remove ambiguity about which text is part of the variable name, you can surround the variable name in curly braces:

```
    path => "${apache::root}/${apache::vhostdir}/${name}",
```

EXPRESSION INTERPOLATION

> Note: This is not recommended.

In a double-quoted string, you may interpolate the value of an arbitrary expression (which may contain both variables and literal values) by putting it inside `${}` (a pair of curly braces preceded by a dollar sign):

```
    file {'config.yml':
      content => "...
db_remote: ${ $clientcert !~ /^db\d+/ }
...",
      ensure => file,
    }
```

This is of limited use, since most expressions resolve to boolean or numerical values.

Behavioral oddities of interpolated expressions:

- You may not use bare word strings or numbers; all literal string or number values must be quoted. The behavior of bare words in an interpolated expression is undefined.
- Within the `${}`, you may use double or single quotes without needing to escape them.

- Interpolated expressions may not use function calls as operands.

ESCAPE SEQUENCES

The following escape sequences are available:

- `\$` — literal dollar sign
- `\"` — literal double quote
- `\'` — literal single quote
- `\\` — single backslash
- `\n` — newline
- `\r` — carriage return (version 2.7.20 and higher)
- `\t` — tab
- `\s` — space

Line Breaks

Quoted strings may continue over multiple lines, and line breaks are preserved as a literal part of the string.

Puppet does not attempt to convert line breaks, which means that the type of line break (Unix/LF or Windows/CRLF) used in the file will be preserved.

- To insert an LF in a manifest file saved with Windows line endings, you can use the `\n` escape sequence.
- Puppet ≥ 2.7.20 only: To insert a CRLF in a manifest file saved with Unix line endings, you can use `\r\n`.

Since Puppet 2.7.19 and earlier do not support the `\r` escape sequence, they have no good way to insert a literal CRLF in a manifest file saved with Unix line endings. It is possible to mix-and-match line endings in a single file, but most text editors do not handle it gracefully and will "help" you enough to render it impractical.

Encoding

Puppet treats strings as sequences of bytes. It does not recognize encodings or translate between them, and non-printing characters are preserved.

However, Puppet Labs recommends that all strings be valid UTF8. Future versions of Puppet may impose restrictions on string encoding, and using only UTF8 will protect you in this event. Additionally, PuppetDB will remove invalid UTF8 characters when storing catalogs.

Resource References

Resource references identify a specific existing Puppet resource by its type and title. Several attributes, such as the relationship metaparameters, require resource references.

```
    # A reference to a file resource:
    subscribe => File['/etc/ntp.conf'],
    ...
    # A type with a multi-segment name:
    before => Concat::Fragment['apache_port_header'],
```

The general form of a resource reference is:

- The resource type, capitalized (every segment must be capitalized if the type includes a namespace separator [`::`])
- An opening square bracket
- The title of the resource, or a comma-separated list of titles
- A closing square bracket

Unlike variables, resource references are not parse-order dependent, and can be used before the resource itself is declared.

Multi-Resource References

Resource references with an array of titles or comma-separated list of titles refer to multiple resources of the same type:

```
    # A multi-resource reference:
    require => File['/etc/apache2/httpd.conf', '/etc/apache2/magic',
'/etc/apache2/mime.types'],
    # An equivalent multi-resource reference:
    $my_files = ['/etc/apache2/httpd.conf', '/etc/apache2/magic',
'/etc/apache2/mime.types']
    require => File[$my_files]
```

They can be used wherever an array of references might be used. They can also go on either side of a chaining arrow or receive a block of additional attributes.

Numbers

Puppet's arithmetic expressions accept integers and floating point numbers. Internally, Puppet treats numbers like strings until they are used in a numeric context.

Numbers can be written as bare words or quoted strings, and may consist only of digits with an optional negative sign (`-`) and decimal point.

```
    $some_number = 8 * -7.992
```

```
$another_number = $some_number / 4
```

Numbers cannot include explicit positive signs (`+`) or exponents. Numbers between -1 and 1 cannot start with a bare decimal point; they must have a leading zero.

```
$product = 8 * +4 # syntax error
$product = 8 * 4 # OK
$product = 8 * .12 # syntax error
$product = 8 * 0.12 # OK
```

Arrays

Arrays are written as comma-separated lists of items surrounded by square brackets. An optional trailing comma is allowed between the final value and the closing square bracket.□

```
[ 'one', 'two', 'three' ]
# Equivalent:
[ 'one', 'two', 'three', ]
```

The items in an array can be any data type, including hashes or more arrays.

Resource attributes which can optionally accept multiple values (including the relationship metaparameters) expect those values in an array.

Indexing

You can access items in an array by their numerical index (counting from zero). Square brackets are used for indexing.

Example:

```
$foo = [ 'one', 'two', 'three' ]
notice( $foo[1] )
```

This manifest would log `two` as a notice. (`$foo[0]` would be `one`, since indexing counts from zero.)

Nested arrays and hashes can be accessed by chaining indexes:

```
$foo = [ 'one', {'second' => 'two', 'third' => 'three'} ]
notice( $foo[1]['third'] )
```

This manifest would log `three` as a notice. (`$foo[1]` is a hash, and we access a key named `'third'`.)

Arrays support negative indexing, with `-1` being the final element of the array:

```
$foo = [ 'one', 'two', 'three', 'four', 'five' ]
notice( $foo[2] )
notice( $foo[-2] )
```

The first notice would log `three`, and the second would log `four`.

Additional Functions

The puppetlabs-stdlib module contains several additional functions for dealing with arrays, including:

- `delete`
- `delete_at`
- `flatten`
- `grep`
- `hash`
- `is_array`
- `join`
- `member`
- `prefix`
- `range`
- `reverse`
- `shuffle`
- `size`
- `sort`
- `unique`
- `validate_array`
- `values_at`
- `zip`

Significant Bugs: Mutability

Due to a bug in Puppet, arrays are mutable — their contents can be changed within a given scope. New elements can be added by assigning a value to a previously unused index (`$myarray[6] = "New value"`) or re-assigning a value to an existing index.

This behavior is considered a bug; do not use it. It will not be removed during the Puppet 2.7 series, but may be removed thereafter without a deprecation period.

Hashes

Hashes are written as key/value pairs surrounded by curly braces; a key is separated from its value by a `=>` (arrow, fat comma, or hash rocket), and adjacent pairs are separated by commas. An optional trailing comma is allowed between the final value and the closing curly brace.

```
{ key1 => 'val1', key2 => 'val2' }
# Equivalent:
{ key1 => 'val1', key2 => 'val2', }
```

Hash keys are strings, but hash values can be any data type, including arrays or more hashes.

Indexing

You can access hash members with their key; square brackets are used for indexing.

```
$myhash = { key       => "some value",
            other_key => "some other value" }
notice( $myhash[key] )
```

This manifest would log `some value` as a notice.

Nested arrays and hashes can be accessed by chaining indexes:

```
$main_site = { port        => { http  => 80,
                                https => 443 },
               vhost_name  => 'docs.puppetlabs.com',
               server_name => { mirror0 => 'warbler.example.com',
                                mirror1 => 'egret.example.com' }
             }
notice ( $main_site[port][https] )
```

This example manifest would log `443` as a notice.

Additional Functions

The puppetlabs-stdlib module contains several additional functions for dealing with hashes, including:

- `has_key`
- `is_hash`
- `keys`

- `merge`
- `validate_hash`
- `values`

Significant Bugs: Mutability

Due to a bug in Puppet, hashes are mutable — their contents can be changed within a given scope. New elements can be added by assigning a value to a previously unused key (`$myhash[new_key] = "New value"`), although existing keys cannot be reassigned.

This behavior is considered a bug; do not use it. It will not be removed during the Puppet 2.7 series, but may be removed thereafter without a deprecation period.

Regular Expressions

Regular expressions (regexes) are Puppet's one non-standard data type. They cannot be assigned to variables, and they can only be used in the few places that specifically accept regular expressions. These places include: the `=~` and `!~` regex match operators, the cases in selectors and case statements, and the names of node definitions. They cannot be passed to functions or used in resource attributes. (Note that the `regsubst` function takes a stringified regex in order to get around this.)

Regular expressions are written as standard Ruby regular expressions (valid for the version of Ruby being used by Puppet) and must be surrounded by forward slashes:

```
if $host =~ /^www(\d+)\./ {
  notify { "Welcome web server #$1": }
}
```

Alternate forms of regex quoting are not allowed and Ruby-style variable interpolation is not available.

Regex Options

Regexes in Puppet cannot have options or encodings appended after the final slash. However, you may turn options on or off for portions of the expression using the `(?<ENABLED OPTION>:<SUBPATTERN>)` and `(?-<DISABLED OPTION>:<SUBPATTERN>)` notation. The following example enables the `i` option while disabling the `m` and `x` options:

```
$packages = $operatingsystem ? {
  /(?i-mx:ubuntu|debian)/        => 'apache2',
```

```
    /(?i-mx:centos|fedora|redhat)/ => 'httpd',
  }
```

The following options are allowed:

- i — Ignore case
- m — Treat a newline as a character matched by `.`
- x — Ignore whitespace and comments in the pattern

Regex Capture Variables

Within conditional statements that use regexes (but not node definitions that use them), any captures from parentheses in the pattern will be available inside the associated value as numbered variables (`$1, $2`, etc.), and the entire match will be available as `$0`.

These are not normal variables, and have some special behaviors:

- The values of the numbered variables do not persist outside the code block associated with the pattern that set them.
- In nested conditionals, each conditional has its own set of values for the set of numbered variables. At the end of an interior statement, the numbered variables are reset to their previous values for the remainder of the outside statement. (This causes conditional statements to act like [local scopes][local], but only with regard to the numbered variables.)

Language: Comments

Puppet supports two types of comments:

Shell-Style Comments

Shell-style comments (also known as Ruby-style comments) begin with a hash symbol (`#`) and continue to the end of a line. They can start at the beginning of a line or partway through a line that began with code.

```
# This is a comment
file {'/etc/ntp.conf': # This is another comment
  ensure => file,
  owner  => root,
}
```

C-Style Comments

C-style comments are delimited by slashes with inner asterisks. They can span multiple lines. This comment style is less frequently used than shell-style.

```
/*
  this is a comment
*/
```

Language: Virtual Resources

A virtual resource declaration specifies a desired state for a resource without adding it to the catalog. You can then add the resource to the catalog by realizing it elsewhere in your manifests. This splits the work done by a normal resource declaration into two steps.

Although virtual resources can only be declared once, they can be realized any number of times (much as a class may be `included` multiple times).

Purpose

Virtual resources are useful for:

- Resources whose management depends on at least one of multiple conditions being met
- Overlapping sets of resources which may be required by any number of classes
- Resources which should only be managed if multiple cross-class conditions are met

Virtual resources can be used in some of the same situations as classes, since they both offer a safe way to add a resource to the catalog in more than one place. The features that distinguish virtual resources are:

- Searchability via resource collectors, which lets you realize overlapping clumps of virtual resources
- Flatness, such that you can declare a virtual resource and realize it a few lines later without having to clutter your modules with many single-resource classes

For more details, see Virtual Resource Design Patterns.

Syntax

Virtual resources are used in two steps: declaring and realizing.

```
# <modulepath>/apache/manifests/init.pp
...
# Declare:
@a2mod { 'rewrite':
  ensure => present,
} # note: The a2mod type is from the puppetlabs-apache module.

# <modulepath>/wordpress/manifests/init.pp
```

```
...
# Realize:
realize A2mod['rewrite']

# <modulepath>/freight/manifests/init.pp
...
# Realize again:
realize A2mod['rewrite']
```

In the example above, the `apache` class declares a virtual resource, and both the `wordpress` and `freight` classes realize it. The resource will be managed on any node that has the `wordpress` and/or `freight` classes applied to it.

Declaring a Virtual Resource

To declare a virtual resource, prepend `@` (the "at" sign) to the type of a normal resource declaration:

```
@user {'deploy':
  uid     => 2004,
  comment => 'Deployment User',
  group   => www-data,
  groups  => ["enterprise"],
  tag     => [deploy, web],
}
```

Realizing With the `realize` Function

To realize one or more virtual resources by title, use the `realize` function, which accepts one or more resource references:

```
realize User['deploy'], User['zleslie']
```

The `realize` function may be used multiple times on the same virtual resource and the resource will only be added to the catalog once.

Realizing With a Collector

Any resource collector will realize any virtual resource that matches its search expression:

```
User <| tag == web |>
```

You can use multiple resource collectors that match a given virtual resource and it will only be added to the catalog once.

Note that a collector used in an override block or a chaining statement will also realize any matching virtual resources.

Behavior

By itself, a virtual resource declaration will not add any resources to the catalog. Instead, it makes the virtual resource available to the compiler, which may or may not realize it. A matching resource collector or a call to the `realize` function will cause the compiler to add the resource to the catalog.

Parse-Order Independence

Virtual resources do not depend on parse order. You may realize a virtual resource before the resource has been declared.

Collectors vs. the `realize` Function

The `realize` function will cause a compilation failure if you attempt to realize a virtual resource that has not been declared. Resource collectors will fail silently if they do not match any resources.

Virtual Resources in Classes

If a virtual resource is contained in a class, it cannot be realized unless the class is declared at some point during the compilation. A common pattern is to declare a class full of virtual resources and then use a collector to choose the set of resources you need:

```
include virtual::users
User <| groups == admin or group == wheel |>
```

Defined Resource Types

You may declare virtual resources of defined resource types. This will cause every resource contained in the defined resource to behave virtually — they will not be added to the catalog unless the defined resource is realized.

Language: Exported Resources

Note: Exported resources require resource stashing (AKA “storeconfigs”) to be enabled on your puppet master. Resource stashing is provided by PuppetDB. To enable resource stashing, follow these instructions:

- Install PuppetDB on a server at your site
- Connect your puppet master to PuppetDB

(Resource stashing may also be provided by the legacy `active_record` storeconfigs backend. However, all new users should avoid it and use PuppetDB instead.)

An exported resource declaration specifies a desired state for a resource, does not manage the resource on the target system, and publishes the resource for use by other nodes. Any node (including the node that exported it) can then collect the exported resource and manage its own copy of it.

Purpose

Exported resources allow nodes to share information with each other. This is useful when one node has information that another node needs in order to manage a resource — the node with the information can construct and publish the resource, and the node managing the resource can collect it.

The most common use cases are monitoring and backups. A class that manages a service like PostgreSQL can export a `nagios_service` resource describing how to monitor the service, including information like its hostname and port. The Nagios server can then collect every `nagios_service` resource, and will automatically start monitoring the Postgres server.

For more details, see Exported Resource Design Patterns.

Syntax

Using exported resources requires two steps: declaring and collecting.

```
class ssh {
  # Declare:
  @@sshkey { $hostname:
    type => dsa,
    key => $sshdsakey,
  }
  # Collect:
  Sshkey <<| |>>
}
```

In the example above, every node with the `ssh` class will export its own SSH host key and then collect the SSH host key of every node (including its own). This will cause every node in the site to trust SSH connections from every other node.

Declaring an Exported Resource

To declare an exported resource, prepend `@@` (a double “at” sign) to the type of a standard resource declaration:

```
@@nagios_service { "check_zfs${hostname}":
  use                 => 'generic-service',
  host_name           => "$fqdn",
  check_command       => 'check_nrpe_1arg!check_zfs',
```

```
    service_description => "check_zfs${hostname}",
    target              => '/etc/nagios3/conf.d/nagios_service.cfg',
    notify              => Service[$nagios::params::nagios_service],
  }
```

Collecting Exported Resources

To collect exported resources you must use an exported resource collector :

```
  Nagios_service <<| |>> # Collect all exported nagios_service resources

  #  Collect exported file fragments for building a Bacula config file:
  Concat::Fragment <<| tag == "bacula-storage-dir-${bacula_director}" |>>
```

(The second example, taken from puppetlabs-bacula, uses the concat module.)

Since any node could be exporting a resource, it is difficult to predict what the title of an exported resource will be. As such, it's usually best to search on a more general attribute. This is one of the main use cases for tags.

See Exported Resource Collectors for more detail on the collector syntax and search expressions.

Behavior

When resource stashing (AKA storeconfigs) is enabled, the puppet master will send a copy of every catalog it compiles to PuppetDB. PuppetDB retains the most recent catalog for every node and provides the puppet master with a search interface to those catalogs.

Declaring an exported resource causes that resource to be added to the catalog and marked with an "exported" flag, which prevents puppet agent from managing the resource (unless it was collected). When PuppetDB receives the catalog, it also takes note of this flag.

Collecting an exported resource causes the puppet master to send a search query to PuppetDB. PuppetDB will respond with every exported resource that matches the search expression, and the puppet master will add those resources to the catalog.

Timing

An exported resource becomes available to other nodes as soon as PuppetDB finishes storing the catalog that contains it. This is a multi-step process and may not happen immediately:

- The puppet master must have compiled a given node's catalog at least once before its resources become available.
- When the puppet master submits a catalog to PuppetDB, it is added to a queue and stored as soon as possible. Depending on the PuppetDB server's workload, there may be a slight delay between a node's catalog being compiled and its resources becoming available.

Uniqueness

Every exported resource must be globally unique across every single node. If two nodes export resources with the same title or same name/namevar and you attempt to collect both, the compilation will fail. (Note: Some pre-1.0 versions of PuppetDB will not fail in this case. This is a bug.)

To ensure uniqueness, every resource you export should include a substring unique to the node exporting it into its title and name/namevar. The most expedient way is to use the `hostname` or `fqdn` facts.

Exported Resource Collectors

Exported resource collectors do not collect normal or virtual resources. In particular, they cannot retrieve non-exported resources from other nodes' catalogs.

Language: Tags

Resources, classes, and defined type instances may have any number of tags associated with them, plus they receive some tags automatically. Tags are useful for:

- Collecting resources
- Analyzing reports
- Restricting catalog runs

Tag Names

See here for the characters allowed in tag names.

Assigning Tags to Resources

A resource may have any number of tags. There are several ways to assign a tag to a resource.

Automatic Tagging

Every resource automatically receives the following tags:

- Its resource type
- The full name of the class and/or defined type in which the resource was declared
- Every namespace segment of the resource's class and/or defined type

For example, a file resource in class `apache::ssl` would get the tags `file`, `apache::ssl`, `apache`, and `ssl`.

Class tags are generally the most useful, especially when setting up tagmail or testing refactored manifests.

Containment

Like relationships and most metaparameters, tags are passed along by containment. This means a resource will receive all of the tags from the class and/or defined type that contains it. In the case of nested containment (e.g. a class that declares a defined resource, or a defined type that declares other defined resources), a resource will receive tags from all of its containers.

The `tag` Metaparameter

You can use the `tag` metaparameter in a resource declaration to add any number of tags:

```
apache::vhost {'docs.puppetlabs.com':
  port => 80,
  tag  => ['us_mirror1', 'us_mirror2'],
}
```

The `tag` metaparameter can accept a single tag or an array. These will be added to the tags the resource already has. Also, `tag` can be used with normal resources, defined resources, and classes (when using the resource-like declaration syntax). Since containment applies to tags, the example above would assign the `us_mirror1` and `us_mirror2` tags to every resource contained by `Apache::Vhost['docs.puppetlabs.com']`.

The `tag` Function

You can use the `tag` function inside a class definition or defined type to assign tags to the surrounding container and all of the resources it contains:

```
class role::public_web {
  tag 'us_mirror1', 'us_mirror2'

  apache::vhost {'docs.puppetlabs.com':
    port => 80,
  }
  ssh::allowgroup {'www-data': }
  @@nagios::website {'docs.puppetlabs.com': }
}
```

The example above would assign the `us_mirror1` and `us_mirror2` tags to all of the defined resources being declared in the class `role::public_web`, as well as to all of the resources each of them contains.

Using Tags

Collecting Resources

Tags can be used as an attribute in the search expression of a resource collector. This is mostly

useful for realizing virtual and exported resources.

Restricting Catalog Runs

Puppet agent and puppet apply can use the `tags` setting to only apply a subset of the node's catalog. This is useful when refactoring modules, and allows you to only apply a single class on a test node.

The `tags` setting can be set in `puppet.conf` (to permanently restrict the catalog) or on the command line (to temporarily restrict it):

```
$ sudo puppet agent --test --tags apache,us_mirror1
```

The value of the `tags` setting should be a comma-separated list of tags (with no spaces between tags).

Sending Tagmail Reports

The built-in tagmail report handler can send emails to arbitrary email addresses whenever resources with certain tags are changed. See the following for more info:

- The tagmail report handler
- The `tagmail.conf` file

Reading Tags in Custom Report Handlers

Resource tags are available to custom report handlers and out-of-band report processors: Each `Puppet::Resource::Status` object and `Puppet::Util::Log` object has a `tags` key whose value is an array containing every tag for the resource in question. See the following pages for more info:

- Processing Reports
- Report Format 2 (the report format used by Puppet 2.7)

Language: Run Stages

Run stages are an additional way to order resources. They allow groups of classes to run before or after nearly everything else, without having to explicitly create relationships with every other class. Run stages were added in Puppet 2.6.0.

Run stages have several major limitations; you should understand these before attempting to use them.

The run stage feature has two parts:

- A `stage` resource type.

- A `stage` metaparameter, which assigns a class to a named run stage.

The Default `main` Stage

By default there is only one stage (named "`main`"). All resources are automatically associated with this stage unless explicitly assigned to a different one. If you do not use run stages, every resource is in the main stage.

Custom Stages

Additional stages are declared as normal resources. Each additional stage must have an order relationship with another stage, such as `Stage['main']`. As with normal resources, these relationships can be specified with metaparameters or with chaining arrows.

```
stage { 'first':
  before => Stage['main'],
}
stage { 'last': }
Stage['main'] -> Stage['last']
```

In the above example, all classes assigned to the `first` stage will be applied before the classes associated with the `main` stage and both stages will be applied before the `last` stage.

Assigning Classes to Stages

Once stages have been declared, a class may be assigned to a custom stage with the `stage` metaparameter.

```
class { 'apt-keys':
  stage => first,
}
```

The above example will ensure that the `apt-keys` class happens before all other classes, which can be useful if most of your package resources rely on those keys.

In order to assign a class to a stage, you must use the resource-like class declaration syntax. You cannot assign classes to stages with the `include` function.

Limitations and Known Issues

- You cannot assign a class to a run stage when declaring it with `include`.
- You cannot subscribe to or notify resources across a stage boundary.
- Due to the "anchor pattern issue" with containment, classes that declare other classes will

behave badly if declared with a run stage. (The second-order classes will “float off” into the main stage, and since the first-order class likely depended on their resources, this will likely cause failures.)

Due to these limitations, stages should only be used with the simplest of classes, and only when absolutely necessary. Mass dependencies like package repositories are effectively the only valid use case.

Language: Importing Manifests

Puppet’s normal behavior is to compile a single manifest (the “site manifest”) and autoload any referenced classes from modules (optionally doing the same with a list of classes from an ENC).

The `import` keyword causes Puppet to compile more than one manifest without autoloading from modules.

> **ASIDE: BEST PRACTICES**
>
> You should generally avoid the `import` keyword. It was introduced to the language before modules existed, and was rendered mostly obsolete once Puppet could autoload classes and defined types from modules. Mixing `import` and modules can often cause bizarre results.
>
> The one modern use for importing is to allow node definitions to be stored in several files. However, note that this requires you to restart the puppet master or edit site.pp whenever you edit your nodes.

Syntax

```
# /etc/puppetlabs/puppet/manifests/site.pp

# import many manifest files with node definitions
import 'nodes/*.pp'

# import a single manifest file with node definitions
import 'nodes.pp'
```

An import statement consists of the `import` keyword, followed by a literal quoted string with no variable interpolation.

The string provided must be a file path or file glob (as implemented by Ruby’s `Dir.glob` method). These paths must resolve to one or more Puppet manifest (.pp) files.

If the file path or glob is not fully qualified, it will be resolved relative to the manifest file in which

the `import` statement is found. Thus, the examples above assume that both the `nodes/` directory and the `nodes.pp` file are in the same `/etc/puppetlabs/puppet/manifests` directory as site.pp.

Behavior

Import statements have the following characteristics:

- They read the contents of the requested file(s) and add their code to top scope
- They are processed before any other code in the manifest is parsed
- They cannot be contained by conditional structures or node/class definitions

These quirks mean the location of an import statement in a manifest does not matter. If an uncommented import statement exists anywhere in a manifest, it will always run (even if it looks like it shouldn't) and the code it imports will not be contained in any definition or conditional. The following example illustrates this:

```
# /etc/puppetlabs/puppet/manifests/site.pp
node 'kestrel.example.com' {
    import 'nodes/kestrel.pp'
}

# /etc/puppetlabs/puppet/manifests/nodes/kestrel.pp
include ntp
include apache2
```

This import statement looks like it should insert code INTO the node definition that contains it; instead, it will insert the code outside any node definition, and it will do so regardless of whether the node definition matches the current node. The `ntp` and `apache2` classes would be applied to every node.

Implications and Best Practices

Due to the non-standard behavior of `import`, any imported file should only contain constructs like node definitions and class definitions, which can exist at top scope without necessarily executing on every node.

Interactions With the Autoloader

The behavior of `import` within autoloaded manifests is undefined, and may vary randomly between minor versions of Puppet. You should never place `import` statements in modules; they should only exist in site.pp.

Inability to Reload

The puppet master service monitors its main site manifest and modules and will reload the files whenever they are edited. However, because it only evaluates file globs when the parent file

containing them is reloaded, it cannot tell when imported manifests have been changed.

Thus, if you use `import` statements, you must manually cause your files to be reloaded whenever you edit your imported manifests. You can do this by:

- Restarting the puppet master
- Editing (or `touch`ing) site.pp to trigger a reload

Module Fundamentals

Puppet Modules

Modules are self-contained bundles of code and data. You can write your own modules or you can download pre-built modules from Puppet Labs' online collection, the Puppet Forge.

Nearly all Puppet manifests belong in modules. The sole exception is the main `site.pp` manifest, which contains site-wide and node-specific code.

Every Puppet user should expect to write at least some of their own modules.

- Continue reading to learn how to write and use Puppet modules.
- See "Installing Modules" for how to install pre-built modules from the Puppet Forge.
- See "Publishing Modules" for how to publish your modules to the Puppet Forge.
- See "Using Plugins" for how to arrange plugins (like custom facts and custom resource types) in modules and sync them to agent nodes.

Using Modules

Modules are how Puppet finds the classes and types it can use — it automatically loads any class or defined type stored in its modules. Within a manifest or from an external node classifier (ENC), any of these classes or types can be declared by name:

```
# /etc/puppetlabs/puppet/site.pp

node default {
  include apache

  class {'ntp':
    enable => false;
  }

  apache::vhost {'personal_site':
    port    => 80,
    docroot => '/var/www/personal',
    options => 'Indexes MultiViews',
  }
}
```

Likewise, Puppet can automatically load plugins (like custom native resource types or custom facts) from modules; see "Using Plugins" for more details.

To make a module available to Puppet, place it in one of the directories in Puppet's modulepath.

The Modulepath

Note: The `modulepath` is a list of directories separated by the system path-separator

character. On 'nix systems, this is the colon (:), while Windows uses the semi-colon (;). The most common default modulepaths are:

- `/etc/puppetlabs/puppet/modules:/opt/puppet/share/puppet/modules` (for Puppet Enterprise)
- `/etc/puppet/modules:/usr/share/puppet/modules` (for open source Puppet)

Use `puppet config print modulepath` to see your currently configured modulepath.

If you want both puppet master and puppet apply to have access to the modules, set the modulepath in puppet.conf to go to the `[main]` block. Modulepath is also one of the settings that can be different per environment.

You can easily install modules written by other users with the `puppet module` subcommand. See "Installing Modules" for details.

Module Layout

On disk, a module is simply a directory tree with a specific, predictable structure:

- MODULE NAME
  - manifests
  - files
  - templates
  - lib
  - tests
  - spec

Example

This example module, named "`my_module`," shows the standard module layout in more detail:

- `my_module` — This outermost directory's name matches the name of the module.
  - `manifests/` — Contains all of the manifests in the module.
    - `init.pp` — Contains a class definition. This class's name must match the module's name.
    - `other_class.pp` — Contains a class named `my_module::other_class`.
    - `my_defined_type.pp` — Contains a defined type named `my_module::my_defined_type`.
    - `implementation/` — This directory's name affects the class names beneath it.
      - `foo.pp` — Contains a class named `my_module::implementation::foo`.
      - `bar.pp` — Contains a class named `my_module::implementation::bar`.
  - `files/` — Contains static files, which managed nodes can download.

        - `service.conf` — This file's URL would be `puppet:///modules/my_module/service.conf`.

    - `lib/` — Contains plugins, like custom facts and custom resource types. See "Using Plugins" for more details.
    - `templates/` — Contains templates, which the module's manifests can use. See "Templates" for more details.
        - `component.erb` — A manifest can render this template with `template('my_module/component.erb')`.

    - `tests/` — Contains examples showing how to declare the module's classes and defined types.
        - `init.pp`
        - `other_class.pp` — Each class or type should have an example in the tests directory.

    - `spec/` — Contains spec tests for any plugins in the lib directory.

Each of the module's subdirectories has a specific function, as follows.

Manifests

Each manifest in a module's `manifests` folder should contain one class or defined type. The file names of manifests map predictably to the names of the classes and defined types they contain.

`init.pp` is special and always contains a class with the same name as the module.

Every other manifest contains a class or defined type named as follows:

Name of module	::	Other directories:: (if any)	Name of file (no extension)
`my_module`	`::`		`other_class`
`my_module`	`::`	`implementation::`	`foo`

Thus:

- `my_module::other_class` would be in the file `my_module/manifests/other_class.pp`
- `my_module::implementation::foo` would be in the file `my_module/manifests/implementation/foo.pp`

The double colon that divides the sections of a class's name is called the namespace separator.

Allowed Module Names

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

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

Certain module names are disallowed:

- main
- settings

Files

Files in a module's `files` directory are automatically served to agent nodes. They can be downloaded by using puppet:/// URLs in the `source` attribute of a `file` resource.

Puppet URLs work transparently in both agent/master mode and standalone mode; in either case, they will retrieve the correct file from a module.

Puppet URLs are formatted as follows:

Protocol	3 slashes	"Modules"/	Name of module/	Name of file
`puppet:`	`///`	`modules/`	`my_module/`	`service.conf`

So `puppet:///modules/my_module/service.conf` would map to `my_module/files/service.conf`.

Templates

Any ERB template (see "Templates" for more details) can be rendered in a manifest with the `template` function. The output of the template is a simple string, which can be used as the content attribute of a `file` resource or as the value of a variable.

The template function can look up templates identified by shorthand:

Template function	('	Name of module/	Name of template	')
`template`	`('`	`my_module/`	`component.erb`	`')`

So `template('my_module/component.erb')` would render the template `my_module/templates/component.erb`.

Writing Modules

To write a module, simply write classes and defined types and place them in properly named manifest files as described above.

- See here for more information on classes
- See here for more information on defined types

Best Practices

The classes, defined types, and plugins in a module should all be related, and the module should aim to be as self-contained as possible.

Manifests in one module should never reference files or templates stored in another module.

Be wary of having classes declare classes from other modules, as this makes modules harder to redistribute. When possible, it's best to isolate "super-classes" that declare many other classes in a local "site" module.

Plugins in Modules

Learn how to distribute custom facts and types from the server to managed clients automatically.

Details

This page describes the deployment of custom facts and types for use by the client via modules.

Custom types and facts are stored in modules. These custom types and facts are then gathered together and distributed via a file mount on your Puppet master called plugins.

This technique can also be used to bundle functions for use by the server when the manifest is being compiled. Doing so is a two step process which is described further on in this document.

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

Note: Plugins in modules is supported in 0.24.x onwards and modifies the pluginsync model supported in releases prior to 0.24.x. It is NOT supported in earlier releases of Puppet but may be present as a patch in some older Debian Puppet packages. The older 0.24.x configuration for plugins in modules is documented at the end of this page.

Module structure for 0.25.x and later

In Puppet version 0.25.x and later, plugins are stored in the `lib` directory of a module, using an internal directory structure that mirrors that of the Puppet code:

```
{modulepath}
└── {module}
    └── lib
        ├── augeas
        |   └── lenses
        ├── facter
        └── puppet
            ├── parser
            |   └── functions
            ├── provider
```

```
│   ├── exec
│   ├── package
│   └── etc... (any resource type)
└── type
```

As the directory tree suggests, custom facts should go in `lib/facter/`, custom types should go in `lib/puppet/type/`, custom providers should go in `lib/puppet/provider/{type}/`, and custom functions should go in `lib/puppet/parser/functions/`.

For example:

A custom user provider:

```
{modulepath}/{module}/lib/puppet/provider/user/custom_user.rb
```

A custom package provider:

```
{modulepath}/{module}/lib/puppet/provider/package/custom_pkg.rb
```

A custom type for bare Git repositories:

```
{modulepath}/{module}/lib/puppet/type/gitrepo.rb
```

A custom fact for the root of all home directories (that is, `/home` on Linux, `/Users` on Mac OS X, etc.):

```
{modulepath}/{module}/lib/facter/homeroot.rb
```

A custom Augeas lens:

```
{modulepath}/{module}/lib/augeas/lenses/custom.aug
```

> Note: Support for syncing Augeas lenses was added in Puppet 2.7.18.

And so on.

Most types and facts should be stored in which ever module they are related to; for example, a Bind fact might be distributed in your Bind module. If you wish to centrally deploy types and facts you could create a separate module just for this purpose, for example one called `custom`. This module needs to be a valid module (with the correct directory structure and an `init.pp` file).

So, if we are using our custom module and our modulepath is /etc/puppet/modules then types and facts would be stored in the following directories:

```
/etc/puppet/modules/custom/lib/puppet/type
/etc/puppet/modules/custom/lib/puppet/provider
/etc/puppet/modules/custom/lib/puppet/parser/functions
/etc/puppet/modules/custom/lib/facter
```

Note: 0.25.x versions of Puppet have a known bug whereby plugins are instead loaded from the deprecated `plugins` directories of modules when applying a manifest locally with the `puppet` command, even though puppetmasterd will correctly serve the contents of `lib/` directories to agent nodes. This bug is fixed in Puppet 2.6.

Enabling Pluginsync

After setting up the directory structure, we then need to turn on pluginsync in our puppet.conf configuration file on both the master and the clients:

```
[main]
pluginsync = true
```

Note on Usage for Server Custom Functions

Functions are executed on the server while compiling the manifest. A module defined in the manifest can include functions in the plugins directory. The custom function will need to be placed in the proper location within the manifest first:

```
{modulepath}/{module}/lib/puppet/parser/functions
```

Note that this location is not within the puppetmaster's $libdir path. Placing the custom function within the module plugins directory will not result in the puppetmasterd loading the new custom function. The puppet client can be used to help deploy the custom function by copying it from modulepath/module/lib/puppet/parser/functions to the proper $libdir location. To do so run the puppet client on the server. When the client runs it will download the custom function from the module's lib directory and deposit it within the correct location in $libdir. The next invocation of the Puppet master by a client will autoload the custom function.

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

Legacy 0.24.x and Plugins in Modules

For older Puppet release the `lib` directory was called `plugins`.

So for types you would place them in:

```
{modulepath}/{module}/plugins/puppet/type
```

For providers you place them in:

```
{modulepath}/{module}/plugins/puppet/provider
```

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

```
{modulepath}/{module}/plugins/facter
```

If we are using our custom module and our modulepath is /etc/puppet/modules then types and facts would be stored in the following directories:

```
/etc/puppet/modules/custom/plugins/puppet/type
/etc/puppet/modules/custom/plugins/puppet/provider
/etc/puppet/modules/custom/plugins/facter
```

Enabling pluginsync for 0.24.x versions

For 0.24.x versions you may need to specify some additional options:

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

Installing Modules

Installing Modules

This reference applies to Puppet 2.7.14 and later and Puppet Enterprise 2.5 and later. Earlier versions will not behave identically.

The puppet module tool does not currently work on Windows.

- Windows nodes which pull configurations from a Linux or Unix puppet master can use any Forge modules installed on the master. Continue reading to learn how to use the module tool on your puppet master.
- On Windows nodes which compile their own catalogs, you can install a Forge module by downloading and extracting the module's release tarball, renaming the module directory to remove the user name prefix, and moving it into place in Puppet's modulepath.

The Puppet Forge is a repository of pre-existing modules, written and contributed by users. These modules solve a wide variety of problems so using them can save you time and effort.

The `puppet module` subcommand, which ships with Puppet, is a tool for finding and managing new modules from the Forge. Its interface is similar to several common package managers, and makes it easy to search for and install new modules from the command line.

- Continue reading to learn how to install and manage modules from the Puppet Forge.
- See "Module Fundamentals" to learn how to use and write Puppet modules.
- See "Publishing Modules" to learn how to contribute your own modules to the Forge, including information about the puppet module tool's `build` and `generate` actions.
- See "Using Plugins" for how to arrange plugins (like custom facts and custom resource types) in modules and sync them to agent nodes.

Using the Module Tool

The `puppet module` subcommand has several actions. The main actions used for managing modules are:

`install`

Install a module from the Forge or a release archive.

```
# puppet module install puppetlabs-apache --version 0.0.2
```

`list`

List installed modules.

```
# puppet module list
```

`search`

Search the Forge for a module.

```
# puppet module search apache
```

`uninstall`

Uninstall a puppet module.

```
# puppet module uninstall puppetlabs-apache
```

`upgrade`

Upgrade a puppet module.

```
# puppet module upgrade puppetlabs-apache --version 0.0.3
```

If you have used a command line package manager tool (like `gem`, `apt-get`, or `yum`) before, these actions will generally do what you expect. You can view a full description of each action with `puppet man module` or by viewing the man page here.

Installing Modules

The `puppet module install` action will install a module and all of its dependencies. By default, it will install into the first directory in Puppet's modulepath.

- Use the `--version` option to specify a version. You can use an exact version or a requirement string like `>=1.0.3`.
- Use the `--force` option to forcibly re-install an existing module.
- Use the `--environment` option to install into a different environment.
- Use the `--modulepath` option to manually specify which directory to install into. Note: To avoid duplicating modules installed as dependencies, you may need to specify the modulepath as a list of directories; see the documentation for setting the modulepath for details.
- Use the `--ignore-dependencies` option to skip installing any modules required by this module.

Installing From the Puppet Forge

To install a module from the Puppet Forge, simply identify the desired module by its full name. The full name of a Forge module is formatted as "username-modulename."

```
# puppet module install puppetlabs-apache
```

Installing From Another Module Repository

The module tool can install modules from other repositories that mimic the Forge's interface. To do this, change the `module_repository` setting in `puppet.conf` or specify a repository on the command line with the `--module_repository` option. The value of this setting should be the base URL of the repository; the default value, which uses the Forge, is `http://forge.puppetlabs.com`.

After setting the repository, follow the instructions above for installing from the Forge.

```
# puppet module install --module_repository http://dev-forge.example.com
puppetlabs-apache
```

Installing From a Release Tarball

At this time, the module subcommand cannot properly install from local tarball files. Follow issue #13542 for more details about the progress of this feature.

Finding Modules

Modules can be found by browsing the Forge's web interface or by using the module tool's `search` action. The search action accepts a single search term and returns a list of modules whose names, descriptions, or keywords match the search term.

```
$ puppet module search apache
Searching http://forge.puppetlabs.com ...
NAME                          DESCRIPTION            AUTHOR          KEYWORDS
puppetlabs-apache             This is a generic ...  @puppetlabs     apache
web
puppetlabs-passenger          Module to manage P...  @puppetlabs     apache
DavidSchmitt-apache           Manages apache, mo...  @DavidSchmitt   apache
jamtur01-httpauth             Puppet HTTP Authen...  @jamtur01       apache
jamtur01-apachemodules        Puppet Apache Modu...  @jamtur01       apache
adobe-hadoop                  Puppet module to d...  @adobe          apache
adobe-hbase                   Puppet module to d...  @adobe          apache
adobe-zookeeper               Puppet module to d...  @adobe          apache
adobe-highavailability        Puppet module to c...  @adobe          apache
mon
adobe-mon                     Puppet module to d...  @adobe          apache
mon
puppetmanaged-webserver       Apache webserver m...  @puppetmanaged  apache
ghoneycutt-apache             Manages apache ser...  @ghoneycutt     apache
web
ghoneycutt-sites              This module manage...  @ghoneycutt     apache
web
fliplap-apache_modules_sles11 Exactly the same a...  @fliplap
mstanislav-puppet_yum         Puppet 2.              @mstanislav     apache
mstanislav-apache_yum         Puppet 2.              @mstanislav     apache
jonhadfield-wordpress         Puppet module to s...  @jonhadfield    apache
```

```
php
saz-php                         Manage cli, apache...  @saz            apache
php
pmtacceptance-apache            This is a dummy ap...  @pmtacceptance  apache
php
pmtacceptance-php               This is a dummy ph...  @pmtacceptance  apache
php
```

Once you've identified the module you need, you can install it by name as described above.

Managing Modules

Listing Installed Modules

Use the module tool's `list` action to see which modules you have installed (and which directory they're installed in).

- Use the `--tree` option to view the modules arranged by dependency instead of by location on disk.

Upgrading Modules

Use the module tool's `upgrade` action to upgrade an installed module to the latest version. The target module must be identified by its full name.

- Use the `--version` option to specify a version.
- Use the `--ignore-dependencies` option to skip upgrading any modules required by this module.

Uninstalling Modules

Use the module tool's `uninstall` action to remove an installed module. The target module must be identified by its full name:

```
# puppet module uninstall apache
Error: Could not uninstall module 'apache':
  Module 'apache' is not installed
      You may have meant `puppet module uninstall puppetlabs-apache`
# puppet module uninstall puppetlabs-apache
Removed /etc/puppet/modules/apache (v0.0.3)
```

By default, the tool won't uninstall a module which other modules depend on or whose files have been edited since it was installed.

- Use the `--force` option to uninstall even if the module is depended on or has been manually edited.

Publishing Modules on the Puppet Forge

The Puppet Forge is a repository of modules, written and contributed by users. This document describes how to publish your own modules to the Puppet Forge so that other users can install them.

- Continue reading to learn how to publish your modules to the Puppet Forge.
- See "Module Fundamentals" for how to write and use your own Puppet modules.
- See "Installing Modules" for how to install pre-built modules from the Puppet Forge.
- See "Using Plugins" for how to arrange plugins (like custom facts and custom resource types) in modules and sync them to agent nodes.

Overview

This guide assumes that you have already written a useful Puppet module. To publish your module, you will need to:

1. Create a Puppet Forge account, if you don't already have one
2. Prepare your module
3. Write a Modulefile with the required metadata
4. Build an uploadable tarball of your module
5. Upload your module using the Puppet Forge's web interface.

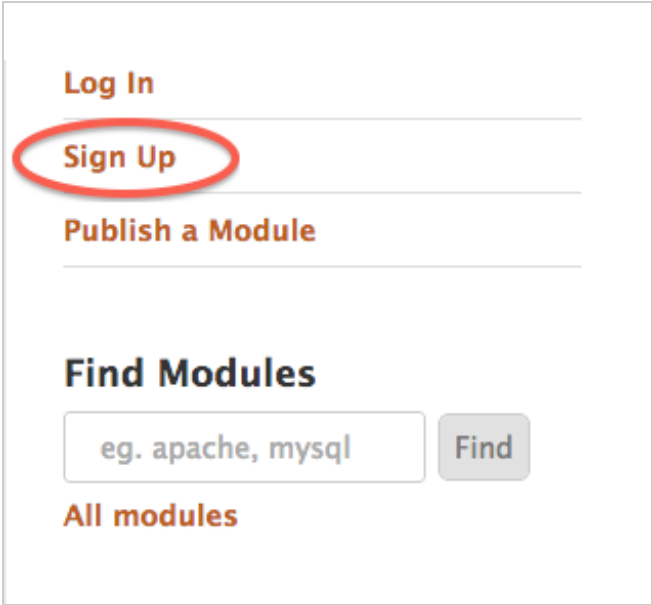
> **A Note on Module Names**
>
> Because many users have published their own versions of modules with common names ("mysql," "bacula," etc.), the Puppet Forge requires module names to have a username prefix. That is, if a user named "puppetlabs" maintained a "mysql" module, it would be known to the Puppet Forge as `puppetlabs-mysql`.
>
> Be sure to use this long name in your module's Modulefile. However, you do not have to rename the module's directory, and can leave the module in your active modulepath — the build action will do the right thing as long as the Modulefile is correct.

> **Another Note on Module Names**
>
> Although the Puppet Forge expects to receive modules named `username-module`, its web interface presents them as `username/module`. There isn't a good reason for this, and we are working on reconciling the two; in the meantime, be sure to always use the `username-module` style in your metadata files and when issuing commands.

Create a Puppet Forge Account

Before you begin, you should create a user account on the Puppet Forge. You will need to know your username when preparing to publish any of your modules.

Start by navigating to the Puppet Forge website and clicking the “Sign Up” link in the sidebar:



Fill in your details. After you finish, you will be asked to verify your email address via a verification email. Once you have done so, you can publish modules to the Puppet Forge.

Prepare the Module

If you already have a Puppet module with the correct directory layout, you may continue to the next step.

Alternately, you can use the `puppet module generate` action to generate a template layout. This is mostly useful if you need an example Modulefile and README, and also includes a copy of the `spec_helper` tool for writing rspec-puppet tests. If you choose to do this, you will need to manually copy your module’s files into the template.

To generate a template, run `puppet module generate <USERNAME>-<MODULE NAME>`. For example:

```
# puppet module generate examplecorp-mymodule
Generating module at /Users/fred/Development/examplecorp-mymodule
examplecorp-mymodule
examplecorp-mymodule/tests
examplecorp-mymodule/tests/init.pp
examplecorp-mymodule/spec
examplecorp-mymodule/spec/spec_helper.rb
examplecorp-mymodule/README
examplecorp-mymodule/Modulefile
examplecorp-mymodule/manifests
examplecorp-mymodule/manifests/init.pp
```

Note: This action is of limited use when developing a module from scratch, as the module must be renamed to remove the username prefix before it can be used with Puppet.

Write a Modulefile

In your module's main directory, create a text file named `Modulefile`. If you generated a template, you'll already have an example Modulefile.

The Modulefile resembles a configuration or data file, but is actually a simple Ruby domain-specific language (DSL), which is executed when you build a tarball of the module. This means Ruby's normal rules of string quoting apply:

```
name 'examplecorp-mymodule'
version '0.0.1'
dependency 'puppetlabs/mysql', '1.2.3'
description "This is a full description
    of the module, and is being written as a multi-line string."
```

Modulefiles support the following pieces of metadata:

- `name` — REQUIRED. The full name of the module, including the username (e.g. "username-module" — see note above).
- `version` — REQUIRED. The current version of the module. This should be a semantic version.
- `summary` — REQUIRED. A one-line description of the module.
- `description` — REQUIRED. A more complete description of the module.
- `dependency` — A module that this module depends on. Unlike the other fields, the `dependency` method accepts up to three comma-separated arguments: a module name (with a slash between the user and name, not a hyphen), a version requirement, and a repository. A Modulefile may include multiple `dependency` lines. See "Dependencies in the Modulefile" below for more details.
- `project_page` — The module's website.
- `license` — The license under which the module is made available.
- `author` — The module's author. If not provided, this field will default to the username portion of the module's `name` field.
- `source` — The module's source. This field's purpose is not specified.

Dependencies in the Modulefile

If you choose to rely on another Forge module, you can express this in the "dependency" field of your Modulefile:

```
dependency 'puppetlabs/stdlib', '>= 2.2.1'
```

> Warning: The full name in a dependency must use a slash between the username and module name. This is different from the name format used elsewhere in the Modulefile. This is a legacy architecture problem with the Puppet Forge, and we apologize for the inconvenience. Our eventual plan is to allow full names with hyphens everywhere while continuing to allow names with slashes, then (eventually, much later) phase out names with slashes.

A Modulefile may have several dependency fields.

The version requirement in a dependency isn't limited to a single version; you can use several operators for version comparisons. The following operators are available:

- `1.2.3` — A specific version.
- `>1.2.3` — Greater than a specific version.
- `<1.2.3` — Less than a specific version.
- `>=1.2.3` — Greater than or equal to a specific version.
- `<=1.2.3` — Less than or equal to a specific version.
- `>=1.0.0 <2.0.0` — Range of versions; both conditions must be satisfied. (This example would match 1.0.1 but not 2.0.1)
- `1.x` — A semantic major version. (This example would match 1.0.1 but not 2.0.1, and is shorthand for `>=1.0.0 <2.0.0`.)
- `1.2.x` — A semantic major & minor version. (This example would match 1.2.3 but not 1.3.0, and is shorthand for `>=1.2.0 <1.3.0`.)

> #### A Note on Semantic Versioning
>
> When writing your Modulefile, you're setting a version for your own module and optionally expressing dependancies on others' module versions. We strongly recommend following the Semantic Versioning specification. Doing so allows others to rely on your modules without unexpected change.
>
> Many other users already use semantic versioning, and you can take advantage of this in your modules' dependencies. For example, if you depend on puppetlabs/stdlib and want to allow updates while avoiding breaking changes, you could write the following line in your Modulefile (assuming a current stdlib version of 2.2.1):
>
> ```
> dependency 'puppetlabs/stdlib', '2.x'
> ```

Build Your Module

Now that the content and Modulefile are ready, you can build a package of your module by running the following command:

```
puppet module build <MODULE DIRECTORY>
```

This will generate a `.tar.gz` package, which will be saved in the module's `pkg/` subdirectory.

For example:

```
# puppet module build /etc/puppetlabs/puppet/modules/mymodule
Building /etc/puppetlabs/puppet/modules/mymodule for release
/etc/puppetlabs/puppet/modules/mymodule/pkg/examplecorp-mymodule-0.0.1.tar.gz
```

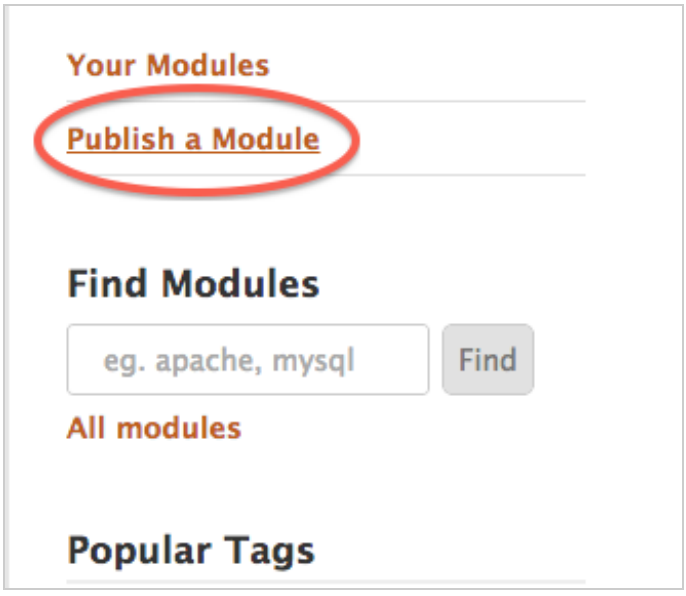
Upload to the Puppet Forge

Now that you have a compiled `tar.gz` package, you can upload it to the Puppet Forge. There is currently no command line tool for publishing; you must use the Puppet Forge's web interface.

In your web browser, navigate to the Puppet Forge; log in if necessary.

Create a Module Page

If you have never published this module before, you must create a new page for it. Click on the "Publish a Module" link in the sidebar:

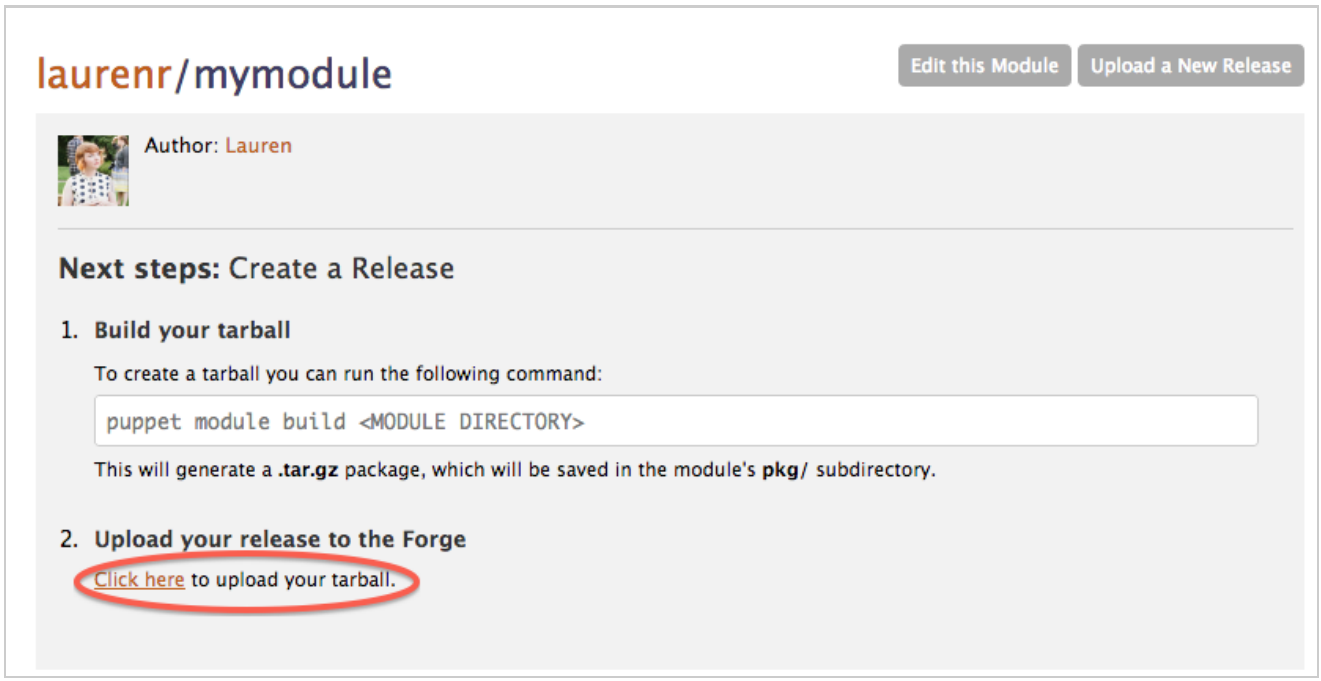


This will bring up a form for info about the new module. Only the "Module Name" field is required. Use the module's short name, not the long `username-module` name.

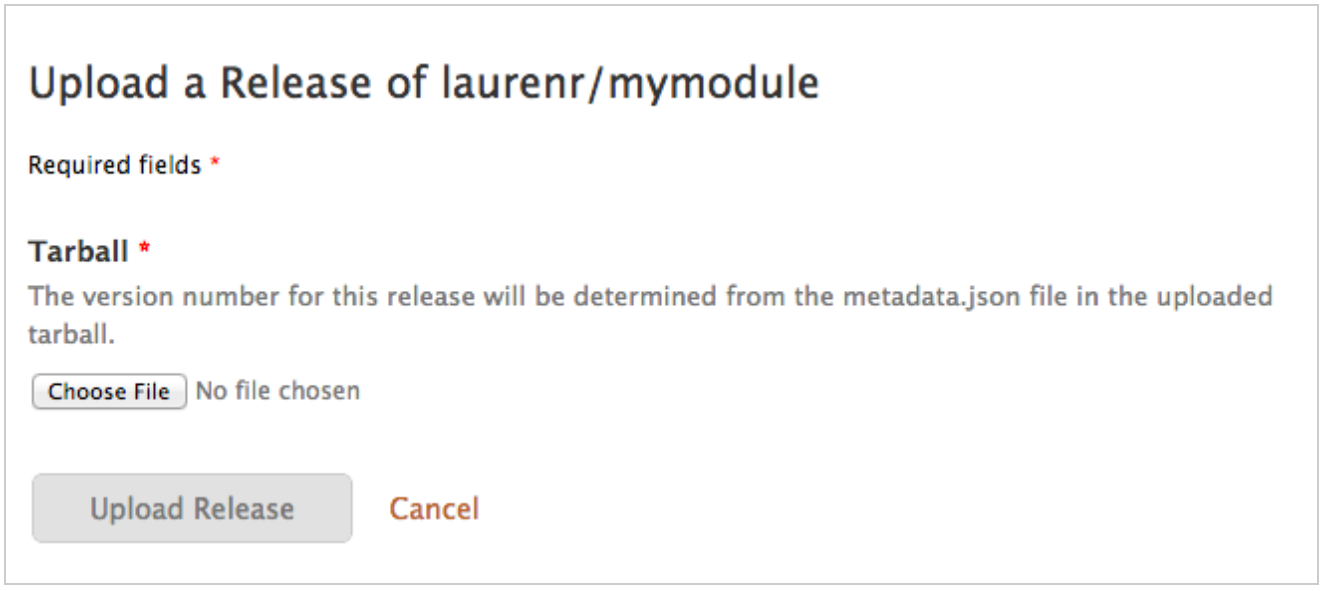
Clicking the “Publish Module” button at the bottom of the form will automatically navigate to the new module page.

Create a Release

Navigate to the module’s page if you are not already there, and click the “Click here to upload your tarball” link:



This will bring you to the upload form:



Click “Choose File” and use the file browser to locate and select the release tarball you created with the `puppet module build` action. Then click the “Upload Release” link.

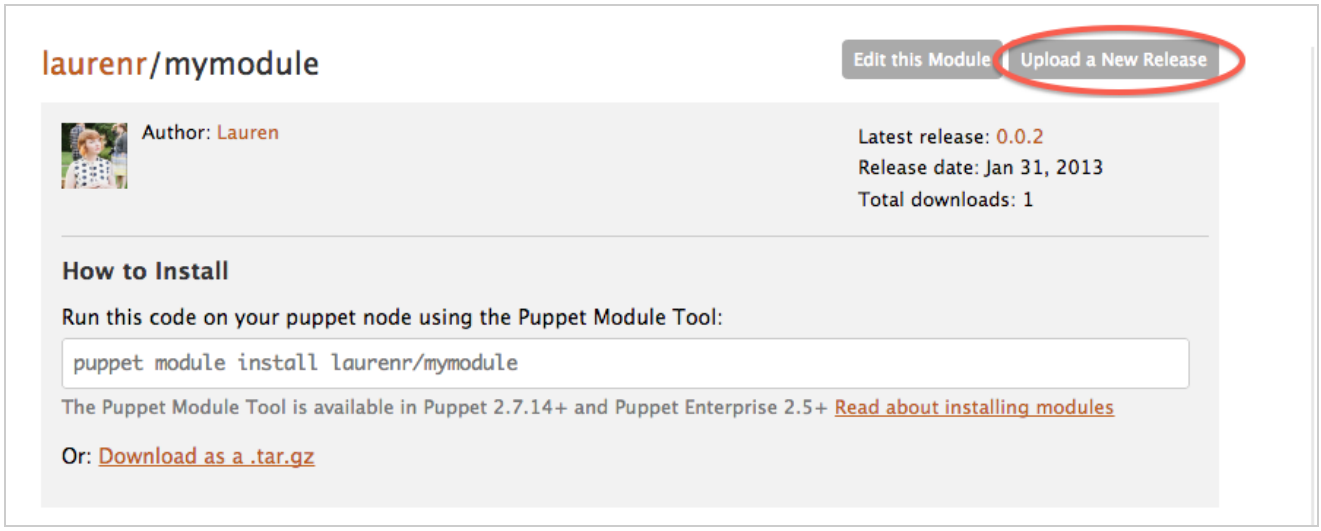
Your module has now been published to the Puppet Forge. The Forge will pull your README, Changelog, and License files from your tarball to display on your module’s page. To confirm that it was published correctly, you can install it on a new system using the `puppet module install`

action.

Release a New Version

To release a new version of an already published module, you will need to make any necessary edits to your module, and then increment the `version` field in the Modulefile (ensuring you use a valid semantic version).

When you are ready to publish your new version, navigate to the Puppet Forge and log in if necessary. Click the "Upload a New Release" link:



This will bring you to the upload form as mentioned in Create a Release above, where you can select the new release tarball and upload the release.

Formats: Reports

The Puppet 2.7 series uses the following report formats:

- 3 (Incorrectly identified in versions 2.7.12 – 2.7.19, correctly identified in versions 2.7.20 and up)
- 2 (versions 2.7.0 – 2.7.11)

Report Format 3

This is the format of reports output by Puppet versions 2.7.12 and later, but the report version identifier did not get changed until Puppet 2.7.20 (See ticket #15739).

Puppet::Transaction::Report

The Puppet::Transaction::Report contains the following attributes:

name	type	description
host	string	the host that generated this report.

time	datetime	when the run began.
logs	array	0 or more Puppet::Util::Log objects.
metrics	hash	maps from string (metric category) to Puppet::Util::Metric.
resource_statuses	hash	maps from resource name to Puppet::Resource::Status
configuration_version	string or integer	The "configuration version" of the puppet run. This is a string if the user has specified their own versioning scheme, otherwise an integer representing seconds since the epoch.
report_format	integer	3
puppet_version	string	The version of the Puppet agent.
kind	string	"inspect" if this report came from a "puppet inspect" run, "apply" if it came from a "puppet apply" or "puppet agent" run.
status	string	"failed", "changed", or "unchanged"
environment	string	The environment that was used for the puppet run.

Puppet::Util::Log

A Puppet::Util::Log object contains the following attributes:

name	type	description
file	string	the pathname of the manifest file which triggered the log message.
line	integer	the line number in the manifest file which triggered the log message.
level	symbol	severity of the message. Possible values for level are :debug, :info, :notice, :warning, :err, :alert, :emerg, :crit
message	string	the message itself.
source	string	the origin of the log message. This could be a resource, a property of a resource, or the string "Puppet".
tags	array	each array element is a string.
time	datetime	when the message was sent.

The `file` and `line` attributes are not always present.

Puppet::Util::Metric

A Puppet::Util::Metric object represents all the metrics in a single category. It contains the following attributes:

name	type	description
name	string	name of the metric category. This is the same as the key associated with this metric in the metrics hash of the Puppet::Transaction::Report.
label	string	This is the "titleized" version of the name, which means underscores are replaced with spaces and the first word is capitalized.
values	array	All the metric values within this category. Each element is of the form [name, titleized_name, value], where name is the name of the particular metric as a string, titleized_name is the "titleized" string of the name, and value is the quantity (an integer or a float).

The set of particular metrics and categories which appear in a report is a fixed set. In a successful report, the categories and metrics are:

- In the `time` category, there is a metric for every resource type for which there is at least one resource in the catalog, plus two additional metrics, called `config_retrieval` and `total`. Each value in the `time` category is a float.
- In the `resources` category, the metrics are `failed`, `out_of_sync`, `changed`, and `total`. Each value in the `resources` category is an integer.
- In the `events` category, there are up to five metrics: `success`, `failure`, `audit`, `noop`, and `total`. `total` is always present; the others are only present when their values are non-zero. Each value in the `events` category is an integer.
- In the `changes` category, there is only one metric, called `total`. Its value is an integer.

Failed reports contain no metrics.

In an inspect report, there is an additional `inspect` metric in the `time` category.

Puppet::Resource::Status

A Puppet::Resource::Status object represents the status of a single resource. It contains the following attributes:

name	type	description
resource_type	string	the resource type, capitalized.
title	title	the resource title.
resource	string	the resource name, in the form Type[title]. This is always the same as the key corresponding to this Puppet::Resource::Status object in the resource_statuses hash. *deprecated*
file	string	the pathname of the manifest file which declared the resource
line	integer	the line number in the manifest file which declared the resource
evaluation_time	float	the amount of time, in seconds, taken to evaluate the resource. Not present in inspect reports.
change_count	integer	the number of properties which changed. Always 0 in inspect reports.
out_of_sync_count	integer	the number of properties which were out of sync. Always 0 in inspect reports.
tags	array	the strings with which the resource is tagged
time	datetime	the time at which the resource was evaluated
events	array	the Puppet::Transaction::Event objects for the resource
out_of_sync	boolean	True if out_of_sync_count > 0, otherwise false. *deprecated*
changed	boolean	True if change_count > 0, otherwise false. *deprecated*
skipped	boolean	True if the resource was skipped, otherwise false.

failed	boolean	True if Puppet experienced an error while evaluating this resource, otherwise false. *deprecated*

Puppet::Transaction::Event

A Puppet::Transaction::Event object represents a single event for a single resource. It contains the following attributes:

name	type	description
audited	boolean	true if this property is being audited, otherwise false. True in inspect reports.
property	string	the property for which the event occurred
previous_value	string, array, or hash	the value of the property before the change (if any) was applied.
desired_value	string, array, or hash	the value specified in the manifest. Absent in inspect reports.
historical_value	string, array, or hash	the audited value from a previous run of Puppet, if known. Otherwise nil. Absent in inspect reports.
message	string	the log message generated by this event
name	symbol	the name of the event. Absent in inspect reports.
status	string	one of the following strings: "success", "failure", "noop", "audit", depending on the type of the event (see below). Always "audit" in inspect reports.
time	datetime	the time at which the property was evaluated

Puppet::Transaction::Event#status has the following meanings:

- `success`: property was out of sync, and was successfully changed to be in sync.
- `failure`: property was out of sync, and couldn't be changed to be in sync due to an error.
- `noop`: property was out of sync, and wasn't changed due to noop mode.
- `audit`: property was in sync, and was being audited.

Differences from Report Format 2□

- Puppet::Transaction::Report gained `environment`.

Report Format 2

This is the format of reports output by Puppet versions 2.6.5 through 2.7.11.

Puppet::Transaction::Report

The Puppet::Transaction::Report contains the following attributes:

name	type	description
host	string	the host that generated this report.
time	datetime	when the run began.

logs	array	0 or more Puppet::Util::Log objects.
metrics	hash	maps from string (metric category) to Puppet::Util::Metric.
resource_statuses	hash	maps from resource name to Puppet::Resource::Status
configuration_version	string or integer	The "configuration version" of the puppet run. This is a string if the user has specified their own versioning scheme, otherwise an integer representing seconds since the epoch.
report_format	integer	2
puppet_version	string	The version of the Puppet agent.
kind	string	"inspect" if this report came from a "puppet inspect" run, "apply" if it came from a "puppet apply" or "puppet agent" run.
status	string	"failed", "changed", or "unchanged"

Puppet::Util::Log

A Puppet::Util::Log object contains the following attributes:

name	type	description
file	string	the pathname of the manifest file which triggered the log message.
line	integer	the line number in the manifest file which triggered the log message.
level	symbol	severity of the message. Possible values for level are :debug, :info, :notice, :warning, :err, :alert, :emerg, :crit
message	string	the message itself.
source	string	the origin of the log message. This could be a resource, a property of a resource, or the string "Puppet".
tags	array	each array element is a string.
time	datetime	when the message was sent.

The “file” and “line” attributes are not always present.

Puppet::Util::Metric

A Puppet::Util::Metric object represents all the metrics in a single category. It contains the following attributes:

name	type	description
name	string	name of the metric category. This is the same as the key associated with this metric in the metrics hash of the Puppet::Transaction::Report.
label	string	This is the "titleized" version of the name, which means underscores are replaced with spaces and the first word is capitalized.
values	array	All the metric values within this category. Each element is of the form [name, titleized_name, value], where name is the name of the particular metric as a string, titleized_name is the "titleized" string of the name, and value is the quantity (an integer or a float).

The set of particular metrics and categories which appear in a report is a fixed set. In a successful report, the categories and metrics are:

- In the “time” category, there is a metric for every resource type for which there is at least one

resource in the catalog, plus two additional metrics, called `config_retrieval` and `total`. Each value in the `time` category is a float.□

- In the "resources" category, the metrics are `failed`, `out_of_sync`, `changed`, and `total`. Each value in the `resources` category is an integer.
- In the `events` category, there are up to five metrics: `success`, `failure`, `audit`, `noop`, and `total`. `total` is always present; the others are only present when their values are non-zero. Each value in the `events` category is an integer.
- In the `changes` category, there is only one metric, called `total`. Its value is an integer.

Failed reports contain no metrics.

In an inspect report, there is an additional `inspect` metric in the `time` category.

Puppet::Resource::Status

A Puppet::Resource::Status object represents the status of a single resource. It contains the following attributes:

name	type	description
resource_type	string	the resource type, capitalized.
title	title	the resource title.
resource	string	the resource name, in the form Type[title]. This is always the same as the key corresponding to this Puppet::Resource::Status object in the resource_statuses hash. *deprecated*
file□	string	the pathname of the manifest file which declared the resource□
line	integer	the line number in the manifest file which declared the resource□
evaluation_time	float□	the amount of time, in seconds, taken to evaluate the resource. Not present in inspect reports.
change_count	integer	the number of properties which changed. Always 0 in inspect reports.
out_of_sync_count	integer	the number of properties which were out of sync. Always 0 in inspect reports.
tags	array	the strings with which the resource is tagged
time	datetime	the time at which the resource was evaluated
events	array	the Puppet::Transaction::Event objects for the resource
out_of_sync	boolean	True if out_of_sync_count > 0, otherwise false. *deprecated*
changed	boolean	True if change_count > 0, otherwise false. *deprecated*
skipped	boolean	True if the resource was skipped, otherwise false.
failed	boolean	True if Puppet experienced an error while evaluating this resource, otherwise false.

Puppet::Transaction::Event

A Puppet::Transaction::Event object represents a single event for a single resource. It contains the

following attributes:

name	type	description
audited	boolean	true if this property is being audited, otherwise false. True in inspect reports.
property	string	the property for which the event occurred
previous_value	string, array, or hash	the value of the property before the change (if any) was applied.
desired_value	string, array, or hash	the value specified in the manifest. Absent in inspect reports.
historical_value	string, array, or hash	the audited value from a previous run of Puppet, if known. Otherwise nil. Absent in inspect reports.
message	string	the log message generated by this event
name	symbol	the name of the event. Absent in inspect reports.
status	string	one of the following strings: "success", "failure", "noop", "audit", depending on the type of the event (see below). Always "audit" in inspect reports.
time	datetime	the time at which the property was evaluated

Puppet::Transaction::Event#status has the following meanings:

- “success”: property was out of sync, and was successfully changed to be in sync.
- “failure”: property was out of sync, and couldn’t be changed to be in sync due to an error.
- “noop”: property was out of sync, and wasn’t changed due to noop mode.
- “audit”: property was in sync, and was being audited.

Differences from Report Format 1□

- Puppet::Transaction::Report gained the properties `configuration_version`, `report_format`, `puppet_version`, `kind`, and `status`. It lost the property `external_times`.
- In Puppet::Resource::Status, `change_count`, `changed`, and `out_of_sync` have new meanings, and `out_of_sync_count` was added. `changed` and `out_of_sync` are now always present. In addition, `resource_type` and `title` were added, and `source_description` and `version` were removed.
- In Puppet::Transaction::Event, the `audited` and `historical_value` properties have been added. The meaning of `previous_value` and `desired_value` has changed. In addition, `file`, `line`, `resource`, `tags`, `source_description`, `version`, and `default_log_level` were removed.
- Metric names are now always strings.
- The metrics changes/total and events/total are now always present.
- The metric time/total has been re-added (it was present in format 0).
- In Puppet::Util::Log, the `version` attribute was removed.

Type Reference

Type Reference

This page is autogenerated; any changes will get overwritten (last generated on Tue Jun 18 17:02:08 -0700 2013)

Resource Types

- The namevar is the parameter used to uniquely identify a type instance. This is the parameter that gets assigned when a string is provided before the colon in a type declaration. In general, only developers will need to worry about which parameter is the `namevar`.

  In the following code:

  ```
  file { "/etc/passwd":
    owner => root,
    group => root,
    mode  => 644
  }
  ```

  `/etc/passwd` is considered the title of the file object (used for things like dependency handling), and because `path` is the namevar for `file`, that string is assigned to the `path` parameter.

- Parameters determine the specific configuration of the instance. They either directly modify the system (internally, these are called properties) or they affect how the instance behaves (e.g., adding a search path for `exec` instances or determining recursion on `file` instances).

- Providers provide low-level functionality for a given resource type. This is usually in the form of calling out to external commands.

  When required binaries are specified for providers, fully qualifed paths indicate that the binary must exist at that specific path and unqualified binaries indicate that Puppet will search for the binary using the shell path.

- Features are abilities that some providers might not support. You can use the list of supported features to determine how a given provider can be used.

  Resource types define features they can use, and providers can be tested to see which features they provide.

---

augeas

Apply a change or an array of changes to the filesystem using the augeas tool.

Requires:

- Augeas

- The ruby-augeas bindings

Sample usage with a string:

```
augeas{"test1" :
  context => "/files/etc/sysconfig/firstboot",
  changes => "set RUN_FIRSTBOOT YES",
  onlyif  => "match other_value size > 0",
}
```

Sample usage with an array and custom lenses:

```
augeas{"jboss_conf":
  context   => "/files",
  changes   => [
      "set etc/jbossas/jbossas.conf/JBOSS_IP $ipaddress",
      "set etc/jbossas/jbossas.conf/JAVA_HOME /usr",
    ],
  load_path => "$/usr/share/jbossas/lenses",
}
```

FEATURES

- execute_changes: Actually make the changes
- need_to_run?: If the command should run
- parse_commands: Parse the command string

Provider	execute changes	need to run?	parse commands
augeas	X	X	X

PARAMETERS

changes

The changes which should be applied to the filesystem. This can be a command or an array of commands. The following commands are supported:

`set <PATH> <VALUE>`

> Sets the value `VALUE` at loction `PATH`

`setm <PATH> <SUB> <VALUE>`

> Sets multiple nodes (matching `SUB` relative to `PATH`) to `VALUE`

`rm <PATH>`

> Removes the node at location `PATH`

`remove <PATH>`

> Synonym for `rm`

`clear <PATH>`

> Sets the node at `PATH` to `NULL`, creating it if needed

`ins <LABEL> (before|after) <PATH>`

> Inserts an empty node `LABEL` either before or after `PATH`.

`insert <LABEL> <WHERE> <PATH>`

> Synonym for `ins`

`mv <PATH> <OTHER PATH>`

> Moves a node at `PATH` to the new location `OTHER PATH`

`move <PATH> <OTHER PATH>`

> Synonym for `mv`

`defvar <NAME> <PATH>`

> Sets Augeas variable `$NAME` to `PATH`

`defnode <NAME> <PATH> <VALUE>`

> Sets Augeas variable `$NAME` to `PATH`, creating it with `VALUE` if needed

If the `context` parameter is set, that value is prepended to any relative `PATH`s.

context

> Optional context path. This value is prepended to the paths of all changes if the path is relative. If the `incl` parameter is set, defaults to `/files + incl`; otherwise, defaults to the empty string.

force

> Optional command to force the augeas type to execute even if it thinks changes will not be made. This does not overide the `onlyif` parameter.

incl

> Load only a specific file, e.g. `/etc/hosts`. This can greatly speed up the execution the

resource. When this parameter is set, you must also set the `lens` parameter to indicate which lens to use.

lens

Use a specific lens, e.g. `Hosts.lns`. When this parameter is set, you must also set the `incl` parameter to indicate which file to load.

load_path

Optional colon-separated list or array of directories; these directories are searched for schema definitions. The agent's `$libdir/augeas/lenses` path will always be added to support pluginsync.

name

The name of this task. Used for uniqueness.

onlyif

Optional augeas command and comparisons to control the execution of this type. Supported onlyif syntax:

- `get <AUGEAS_PATH> <COMPARATOR> <STRING>`
- `match <MATCH_PATH> size <COMPARATOR> <INT>`
- `match <MATCH_PATH> include <STRING>`
- `match <MATCH_PATH> not_include <STRING>`
- `match <MATCH_PATH> == <AN_ARRAY>`
- `match <MATCH_PATH> != <AN_ARRAY>`

where:

- `AUGEAS_PATH` is a valid path scoped by the context
- `MATCH_PATH` is a valid match synatx scoped by the context
- `COMPARATOR` is one of `>, >=, !=, ==, <=,` or `<`
- `STRING` is a string
- `INT` is a number
- `AN_ARRAY` is in the form `['a string', 'another']`

provider

The specific backend to use for this `augeas` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

augeas

Supported features: `execute_changes`, `need_to_run?`, `parse_commands`.

returns

The expected return code from the augeas command. Should not be set.

root

A file system path; all files loaded by Augeas are loaded underneath `root`.

type_check

Whether augeas should perform typechecking. Defaults to false. Valid values are `true`, `false`.

---

computer

Computer object management using DirectoryService on OS X.

Note that these are distinctly different kinds of objects to 'hosts', as they require a MAC address and can have all sorts of policy attached to them.

This provider only manages Computer objects in the local directory service domain, not in remote directories.

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

This type primarily exists to create localhost Computer objects that MCX policy can then be attached to.

Autorequires: If Puppet is managing the plist file representing a Computer object (located at `/var/db/dslocal/nodes/Default/computers/{name}.plist`), the Computer resource will autorequire it.

PARAMETERS

en_address

The MAC address of the primary network interface. Must match en0.

ensure

Control the existences of this computer record. Set this attribute to `present` to ensure the computer record exists. Set it to `absent` to delete any computer records with this name Valid values are `present`, `absent`.

ip_address

The IP Address of the Computer object.

name

The authoritative 'short' name of the computer record.

provider

The specific backend to use for this `computer` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

directoryservice

Computer object management using DirectoryService on OS X. Note that these are distinctly different kinds of objects to 'hosts', as they require a MAC address and can have all sorts of policy attached to them.

This provider only manages Computer objects in the local directory service domain, not in remote directories.

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

Default for `operatingsystem` == `darwin`.

realname

The 'long' name of the computer record.

---

cron

Installs and manages cron jobs. Every cron resource requires a command and user attribute, as well as at least one periodic attribute (hour, minute, month, monthday, weekday, or special). While the name of the cron job is not part of the actual job, it is used by Puppet to store and retrieve it.

If you specify a cron job that matches an existing job in every way except name, then the jobs will be considered equivalent and the new name will be permanently associated with that job. Once this association is made and synced to disk, you can then manage the job normally (e.g., change the schedule of the job).

Example:

```
cron { logrotate:
  command => "/usr/sbin/logrotate",
  user    => root,
  hour    => 2,
  minute  => 0
}
```

Note that all periodic attributes 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 these):

```
cron { logrotate:
  command => "/usr/sbin/logrotate",
  user    => root,
  hour    => ['2-4'],
  minute  => '*/10'
}
```

PARAMETERS

command

> The command to execute in the cron job. The environment provided to the command varies by local system rules, and it is best to always provide a fully qualified command. The user's profile is not sourced when the command is run, so if the user's environment is desired it should be sourced manually.
>
> All cron parameters support `absent` as a value; this will remove any existing values for that field.

ensure

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

environment

> Any environment settings associated with this cron job. They will be stored between the header and the job in the crontab. There can be no guarantees that other, earlier settings will not also affect a given cron job.
>
> Also, Puppet cannot automatically determine whether an existing, unmanaged environment setting is associated with a given cron job. If you already have cron jobs with environment settings, then Puppet will keep those settings in the same place in the file, but will not associate them with a specific job.
>
> Settings should be specified exactly as they should appear in the crontab, e.g., `PATH=/bin:/usr/bin:/usr/sbin`.

hour

> The hour at which to run the cron job. Optional; if specified, must be between 0 and 23, inclusive.

minute

> The minute at which to run the cron job. Optional; if specified, must be between 0 and 59, inclusive.

month

> The month of the year. Optional; if specified must be between 1 and 12 or the month name (e.g., December).

monthday

> The day of the month on which to run the command. Optional; if specified, must be between 1 and 31.

name

> The symbolic name of the cron job. This name is used for human reference only and is generated automatically for cron jobs found on the system. This generally won't matter, as Puppet will do its best to match existing cron jobs against specified jobs (and Puppet adds a comment to cron jobs it adds), but it is at least possible that converting from unmanaged jobs to managed jobs might require manual intervention.

provider

> The specific backend to use for this `cron` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:
>
> **crontab**
>
> > Required binaries: `crontab`.

special

> A special value such as 'reboot' or 'annually'. Only available on supported systems such as Vixie Cron. Overrides more specific time of day/week settings.

target

> Where the cron job should be stored. For crontab-style entries this is the same as the user and defaults that way. Other providers default accordingly.

user

> The user to run the command as. This user must be allowed to run cron jobs, which is not currently checked by Puppet.
>
> The user defaults to whomever Puppet is running as.

weekday

> The weekday on which to run the command. Optional; if specified, must be between 0 and 7, inclusive, with 0 (or 7) being Sunday, or must be the name of the day (e.g., Tuesday).

exec

Executes external commands. It is critical that all commands executed using this mechanism can be run multiple times without harm, i.e., they are idempotent. One useful way to create idempotent commands is to use the checks like `creates` to avoid running the command unless some condition is met.

Note that you can restrict an `exec` to only run when it receives events by using the `refreshonly` parameter; this is a useful way to have your configuration respond to events with arbitrary commands.

Note also that if an `exec` receives an event from another resource, it will get executed again (or execute the command specified in `refresh`, if there is one).

There is a strong tendency to use `exec` to do whatever work Puppet can't already do; while this is obviously acceptable (and unavoidable) in the short term, it is highly recommended to migrate work from `exec` to native Puppet types as quickly as possible. If you find that you are doing a lot of work with `exec`, please at least notify us at Puppet Labs what you are doing, and hopefully we can work with you to get a native resource type for the work you are doing.

Autorequires: If Puppet is managing an exec's cwd or the executable file used in an exec's command, the exec resource will autorequire those files. If Puppet is managing the user that an exec should run as, the exec resource will autorequire that user.

PARAMETERS

command

> (Namevar: If omitted, this parameter's value defaults to the resource's title.)
>
> The actual command to execute. Must either be fully qualified or a search path for the command must be provided. If the command succeeds, any output produced will be logged at the instance's normal log level (usually `notice`), but if the command fails (meaning its return code does not match the specified code) then any output is logged at the `err` log level.

creates

> A file that this command creates. If this parameter is provided, then the command will only be run if the specified file does not exist.
>
> ```
> exec { "tar -xf /Volumes/nfs02/important.tar":
>   cwd     => "/var/tmp",
>   creates => "/var/tmp/myfile",
>   path    => ["/usr/bin", "/usr/sbin"]
> }
> ```

In this example, if `/var/tmp/myfile` is ever deleted, the exec will bring it back by re-extracting the tarball.

cwd

The directory from which to run the command. If this directory does not exist, the command will fail.

environment

Any additional environment variables you want to set for a command. Note that if you use this to set PATH, it will override the `path` attribute. Multiple environment variables should be specified as an array.

group

The group to run the command as. This seems to work quite haphazardly on different platforms – it is a platform issue not a Ruby or Puppet one, since the same variety exists when running commands as different users in the shell.

logoutput

Whether to log output. Defaults to logging output at the loglevel for the `exec` resource. Use on_failure to only log the output when the command reports an error. Values are true, false, on_failure, and any legal log level. Valid values are `true`, `false`, `on_failure`.

onlyif

If this parameter is set, then this `exec` will only run if the command returns 0. For example:

```
exec { "logrotate":
  path   => "/usr/bin:/usr/sbin:/bin",
  onlyif => "test `du /var/log/messages | cut -f1` -gt 100000"
}
```

This would run `logrotate` only if that test returned true.

Note that this command follows the same rules as the main command, which is to say that it must be fully qualified if the path is not set.

Also note that onlyif can take an array as its value, e.g.:

```
onlyif => ["test -f /tmp/file1", "test -f /tmp/file2"]
```

This will only run the exec if all conditions in the array return true.

path

The search path used for command execution. Commands must be fully qualified if no path is specified. Paths can be specified as an array or as a ':' separated list.

provider

The specific backend to use for this `exec` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

posix

Executes external binaries directly, without passing through a shell or performing any interpolation. This is a safer and more predictable way to execute most commands, but prevents the use of globbing and shell built-ins (including control logic like “for” and “if” statements).

Default for `feature` == `posix`.

shell

Passes the provided command through `/bin/sh`; only available on POSIX systems. This allows the use of shell globbing and built-ins, and does not require that the path to a command be fully-qualified. Although this can be more convenient than the `posix` provider, it also means that you need to be more careful with escaping; as ever, with great power comes etc. etc.

This provider closely resembles the behavior of the `exec` type in Puppet 0.25.x.

windows

Execute external binaries on Windows systems. As with the `posix` provider, this provider directly calls the command with the arguments given, without passing it through a shell or performing any interpolation. To use shell built-ins — that is, to emulate the `shell` provider on Windows — a command must explicitly invoke the shell:

```
exec {'echo foo':
  command => 'cmd.exe /c echo "foo"',
}
```

If no extension is specified for a command, Windows will use the `PATHEXT` environment variable to locate the executable.

Note on PowerShell scripts: PowerShell’s default `restricted` execution policy doesn’t allow it to run saved scripts. To run PowerShell scripts, specify the `remotesigned` execution policy as part of the command:

```
exec { 'test':
  path    => 'C:/Windows/System32/WindowsPowerShell/v1.0',
  command => 'powershell -executionpolicy remotesigned -file
C:/test.ps1',
```

```
  }
```

Default for `operatingsystem` == `windows`.

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. The timeout is specified in seconds. The default timeout is 300 seconds and you can set it to 0 to disable the timeout.

tries

The number of times execution of the command should be tried. Defaults to '1'. This many attempts will be made to execute the command until an acceptable return code is returned. Note that the timeout paramater applies to each try rather than to the complete set of tries.

try_sleep

The time to sleep in seconds between 'tries'.

unless

If this parameter is set, then this `exec` will run unless the command returns 0. For example:

```
exec { "/bin/echo root >> /usr/lib/cron/cron.allow":
  path   => "/usr/bin:/usr/sbin:/bin",
  unless => "grep root /usr/lib/cron/cron.allow 2>/dev/null"
}
```

This would add `root` to the cron.allow file (on Solaris) unless `grep` determines it's already there.

Note that this command follows the same rules as the main command, which is to say that it must be fully qualified if the path is not set.

user

The user to run the command as. Note that if you use this then any error output is not currently captured. This is because of a bug within Ruby. If you are using Puppet to create this user, the exec will automatically require the user, as long as it is specified by name.

---

file

Manages files, including their content, ownership, and permissions.

The `file` type can manage normal files, directories, and symlinks; the type should be specified in the `ensure` attribute. Note that symlinks cannot be managed on Windows systems.

File contents can be managed directly with the `content` attribute, or downloaded from a remote source using the `source` attribute; the latter can also be used to recursively serve directories (when the `recurse` attribute is set to `true` or `local`). On Windows, note that file contents are managed in binary mode; Puppet never automatically translates line endings.

Autorequires: If Puppet is managing the user or group that owns a file, the file resource will autorequire them. If Puppet is managing any parent directories of a file, the file resource will autorequire them.

PARAMETERS

backup

Whether files should be backed up before being replaced. The preferred method of backing files up is via a `filebucket`, which stores files by their MD5 sums and allows easy retrieval without littering directories with backups. You can specify a local filebucket or a network-accessible server-based filebucket by setting `backup => bucket-name`. Alternatively, if you specify any value that begins with a `.` (e.g., `.puppet-bak`), then Puppet will use copy the file

in the same directory with that value as the extension of the backup. Setting `backup => false` disables all backups of the file in question.

Puppet automatically creates a local filebucket named `puppet` and defaults to backing up there. To use a server-based filebucket, you must specify one in your configuration.

```
  filebucket { main:
    server => puppet,
    path   => false,
    # The path => false line works around a known issue with the
filebucket type.
  }
```

The `puppet master` daemon creates a filebucket by default, so you can usually back up to your main server with this configuration. Once you've described the bucket in your configuration, you can use it in any file's backup attribute:

```
  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 central 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, you can restore any given file manually (no matter how old), and you can use Puppet Dashboard to view file contents. Eventually, transactional support will be able to automatically restore filebucketed files.

checksum

The checksum type to use when determining whether to replace a file's contents.

The default checksum type is md5. Valid values are `md5`, `md5lite`, `mtime`, `ctime`, `none`.

content

The desired contents of a file, as a string. This attribute is mutually exclusive with `source` and `target`.

Newlines and tabs can be specified in double-quoted strings using standard escaped syntax — \n for a newline, and \t for a tab.

With very small files, you can construct content strings directly in the manifest...

```
  define resolve(nameserver1, nameserver2, domain, search) {
```

```
    $str = "search $search
        domain $domain
        nameserver $nameserver1
        nameserver $nameserver2
        "

    file { "/etc/resolv.conf":
      content => "$str",
    }
}
```

…but for larger files, this attribute is more useful when combined with the template function.

ctime

A read-only state to check the file ctime.

ensure

Whether to create files that don't currently exist. Possible values are absent, present, file, and directory. Specifying `present` will match any form of file existence, and if the file is missing will create an empty file. Specifying `absent` will delete the file (or directory, if `recurse => true`).

Anything other than the above values will create a symlink; note that symlinks cannot be managed on Windows. In the interest of readability and clarity, symlinks should be created by setting `ensure => link` and explicitly specifying a target; however, if a `target` attribute isn't provided, the value of the `ensure` attribute will be used as the symlink target. The following two declarations are equivalent:

```
# (Useful on Solaris)

# Less maintainable:
file { "/etc/inetd.conf":
  ensure => "/etc/inet/inetd.conf",
}

# More maintainable:
file { "/etc/inetd.conf":
  ensure => link,
  target => "/etc/inet/inetd.conf",
}   Valid values are `absent` (also called `false`), `file`, `present`,
`directory`, `link`.  Values can match `/./`.
```

force

Perform the file operation even if it will destroy one or more directories. You must use `force` in order to:

- `purge` subdirectories
- Replace directories with files or links

- Remove a directory when `ensure => absent` Valid values are `true`, `false`.

group

Which group should own the file. Argument can be either a group name or a group ID.

On Windows, a user (such as “Administrator”) can be set as a file’s group and a group (such as “Administrators”) can be set as a file’s owner; however, a file’s owner and group shouldn’t be the same. (If the owner is also the group, files with modes like `0640` will cause log churn, as they will always appear out of sync.)

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

The desired permissions mode for the file, in symbolic or numeric notation. Puppet uses traditional Unix permission schemes and translates them to equivalent permissions for systems which represent permissions differently, including Windows.

Numeric modes should use the standard four-digit octal notation of `<setuid/setgid/sticky><owner><group><other>` (e.g. 0644). Each of the “owner,” “group,” and “other” digits should be a sum of the permissions for that class of users, where read = 4, write = 2, and execute/search = 1. When setting numeric permissions for directories, Puppet sets the search permission wherever the read permission is set.

Symbolic modes should be represented as a string of comma-separated permission clauses, in the form `<who><op><perm>`:

- “Who” should be u (user), g (group), o (other), and/or a (all)
- “Op” should be = (set exact permissions), + (add select permissions), or - (remove select permissions)
- “Perm” should be one or more of:
  - r (read)
  - w (write)
  - x (execute/search)

- t (sticky)
- s (setuid/setgid)
- X (execute/search if directory or if any one user can execute)
- u (user's current permissions)
- g (group's current permissions)
- o (other's current permissions)

Thus, mode `0664` could be represented symbolically as either `a=r,ug+w` or `ug=rw,o=r`. See the manual page for GNU or BSD `chmod` for more details on numeric and symbolic modes.

On Windows, permissions are translated as follows:

- Owner and group names are mapped to Windows SIDs
- The "other" class of users maps to the "Everyone" SID
- The read/write/execute permissions map to the `FILE_GENERIC_READ`, `FILE_GENERIC_WRITE`, and `FILE_GENERIC_EXECUTE` access rights; a file's owner always has the `FULL_CONTROL` right
- "Other" users can't have any permissions a file's group lacks, and its group can't have any permissions its owner lacks; that is, 0644 is an acceptable mode, but 0464 is not.

mtime

A read-only state to check the file mtime.

owner

The user to whom the file should belong. Argument can be a user name or a user ID.

On Windows, a group (such as "Administrators") can be set as a file's owner and a user (such as "Administrator") can be set as a file's group; however, a file's owner and group shouldn't be the same. (If the owner is also the group, files with modes like `0640` will cause log churn, as they will always appear out of sync.)

path

(Namevar: If omitted, this parameter's value defaults to the resource's title.)

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

On Windows, the path should include the drive letter and should use `/` as the separator character (rather than `\\`).

provider

The specific backend to use for this `file` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

posix

Uses POSIX functionality to manage file ownership and permissions.

windows

Uses Microsoft Windows functionality to manage file ownership and permissions.

purge

Whether unmanaged files should be purged. This option only makes sense when managing directories with `recurse => true`.

- When recursively duplicating an entire directory with the `source` attribute, `purge => true` will automatically purge any files that are not in the source directory.
- When managing files in a directory as individual resources, setting `purge => true` will purge any files that aren't being specifically managed.

If you have a filebucket configured, the purged files will be uploaded, but if you do not, this will destroy data. Valid values are `true`, `false`.

recurse

Whether and how deeply to do recursive management. Options are:

- `inf,true` — Regular style recursion on both remote and local directory structure.
- `remote` — Descends recursively into the remote directory but not the local directory. Allows copying of a few files into a directory containing many unmanaged files without scanning all the local files.
- `false` — Default of no recursion.
- `[0-9]+` — Same as true, but limit recursion. Warning: this syntax has been deprecated in favor of the `recurselimit` attribute. Valid values are `true`, `false`, `inf`, `remote`. Values can match `/^[0-9]+$/`.

recurselimit

How deeply to do recursive management. Values can match `/^[0-9]+$/`.

replace

Whether to replace a file that already exists on the local system but whose content doesn't match what the `source` or `content` attribute specifies. Setting this to false allows file resources to initialize files without overwriting future changes. Note that this only affects content; Puppet will still manage ownership and permissions. Defaults to `true`. Valid values are `true` (also called `yes`), `false` (also called `no`).

selinux_ignore_defaults

If this is set then Puppet will not ask SELinux (via matchpathcon) to supply defaults for the

SELinux attributes (seluser, selrole, seltype, and selrange). In general, you should leave this set at its default and only set it to true when you need Puppet to not try to fix SELinux labels automatically. Valid values are `true`, `false`.

selrange

What the SELinux range component of the context of the file should be. Any valid SELinux range component is accepted. For example `s0` or `SystemHigh`. If not specified it defaults to the value returned by matchpathcon for the file, if any exists. Only valid on systems with SELinux support enabled and that have support for MCS (Multi-Category Security).

selrole

What the SELinux role component of the context of the file should be. Any valid SELinux role component is accepted. For example `role_r`. If not specified it defaults to the value returned by matchpathcon for the file, if any exists. Only valid on systems with SELinux support enabled.

seltype

What the SELinux type component of the context of the file should be. Any valid SELinux type component is accepted. For example `tmp_t`. If not specified it defaults to the value returned by matchpathcon for the file, if any exists. Only valid on systems with SELinux support enabled.

seluser

What the SELinux user component of the context of the file should be. Any valid SELinux user component is accepted. For example `user_u`. If not specified it defaults to the value returned by matchpathcon for the file, if any exists. Only valid on systems with SELinux support enabled.

source

A source file, which will be copied into place on the local system. Values can be URIs pointing to remote files, or fully qualified paths to files available on the local system (including files on NFS shares or Windows mapped drives). This attribute is mutually exclusive with `content` and `target`.

The available URI schemes are puppet and file. Puppet URIs will retrieve files from Puppet's built-in file server, and are usually formatted as:

```
puppet:///modules/name_of_module/filename
```

This will fetch a file from a module on the puppet master (or from a local module when using puppet apply). Given a `modulepath` of `/etc/puppetlabs/puppet/modules`, the example above would resolve to `/etc/puppetlabs/puppet/modules/name_of_module/files/filename`.

Unlike `content`, the `source` attribute can be used to recursively copy directories if the

`recurse` attribute is set to `true` or `remote`. If a source directory contains symlinks, use the `links` attribute to specify whether to recreate links or follow them.

Multiple `source` values can be specified as an array, and Puppet will use the first source that exists. This can be used to serve different files to different system types:

```
file { "/etc/nfs.conf":
  source => [
    "puppet:///modules/nfs/conf.$host",
    "puppet:///modules/nfs/conf.$operatingsystem",
    "puppet:///modules/nfs/conf"
  ]
}
```

Alternately, when serving directories recursively, multiple sources can be combined by setting the `sourceselect` attribute to `all`.

sourceselect

Whether to copy all valid sources, or just the first one. This parameter only affects recursive directory copies; by default, the first valid source is the only one used, but if this parameter is set to `all`, then all valid sources will have all of their contents copied to the local system. If a given file exists in more than one source, the version from the earliest source 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. This attribute is mutually exclusive with `source` and `content`.

Symlink targets can be relative, as well as absolute:

```
# (Useful on Solaris)
file { "/etc/inetd.conf":
  ensure => link,
  target => "inet/inetd.conf",
}
```

Directories of symlinks can be served recursively by instead using the `source` attribute, setting `ensure` to `directory`, and setting the `links` attribute to `manage`. Valid values are `notlink`. Values can match `/./`.

type

A read-only state to check the file type.

---

filebucket

A repository for backing up files. If no filebucket is defined, then files will be backed up in their current directory, but the filebucket can be either a host- or site-global repository for backing up. It stores files and returns the MD5 sum, which can later be used to retrieve the file if restoration becomes necessary. A filebucket does not do any work itself; instead, it can be specified as the value of backup in a file object.

Currently, filebuckets are only useful for manual retrieval of accidentally removed files (e.g., you look in the log for the md5 sum and retrieve the file with that sum from the filebucket), but when transactions are fully supported filebuckets will be used to undo transactions.

You will normally want to define a single filebucket for your whole network and then use that as the default backup location:

```
# Define the bucket
filebucket { 'main':
  server => puppet,
  path   => false,
  # Due to a known issue, path must be set to false for remote filebuckets.
}

# Specify it as the default target
File { backup => main }
```

Puppetmaster servers create a filebucket by default, so this will work in a default configuration.

PARAMETERS

name

> The name of the filebucket.

path

> The path to the local filebucket. If this is unset, then the bucket is remote. The parameter server must can be specified to set the remote server.

port

> The port on which the remote server is listening. Defaults to the normal Puppet port, 8140.

server

> The server providing the remote filebucket. If this is not specified then path is checked. If it is set, then the bucket is local. Otherwise the puppetmaster server specified in the config or at the commandline is used.
>
> Due to a known issue, you currently must set the `path` attribute to false if you wish to specify a `server` attribute.

---

group

Manage groups. On most platforms this can only create groups. Group membership must be managed on individual users.

On some platforms such as OS X, group membership is managed as an attribute of the group, not the user record. Providers must have the feature 'manages_members' to manage the 'members' property of a group record.

FEATURES

- manages_aix_lam: The provider can manage AIX Loadable Authentication Module (LAM) system.
- manages_members: For directories where membership is an attribute of groups not users.
- system_groups: The provider allows you to create system groups with lower GIDs.

Provider	manages aix lam	manages members	system groups
aix	X	X	
directoryservice		X	
groupadd			X
ldap			
pw		X	
windows_adsi		X	

PARAMETERS

allowdupe

> Whether to allow duplicate GIDs. Defaults to `false`. Valid values are `true`, `false`.

attribute_membership

> Whether specified attribute value pairs should be treated as the only attributes of the user or whether they should merely be treated as the minimum list. Valid values are `inclusive`, `minimum`.

attributes

> Specify group AIX attributes in an array of `key=value` pairs. Requires features manages_aix_lam.

auth_membership

> whether the provider is authoritative for group membership.

ensure

> Create or remove the group. Valid values are `present`, `absent`.

gid

> The group ID. Must be specified numerically. If no group ID is specified when creating a new group, then one will be chosen automatically according to local system standards. This will likely result in the same group having different GIDs on different systems, which is not recommended.
>
> On Windows, this property is read-only and will return the group's security identifier (SID).

ia_load_module

> The name of the I&A module to use to manage this user Requires features manages_aix_lam.

members

> The members of the group. For directory services where group membership is stored in the group objects, not the users. Requires features manages_members.

name

> The group name. While naming limitations vary by operating system, it is advisable to restrict names to the lowest common denominator, which is a maximum of 8 characters beginning with a letter.
>
> Note that Puppet considers group names to be case-sensitive, regardless of the platform's own rules; be sure to always use the same case when referring to a given group.

provider

> The specific backend to use for this `group` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:
>
> **aix**
>
> > Group management for AIX.
> >
> > Required binaries: `/usr/bin/chgroup`, `/usr/sbin/lsgroup`, `/usr/sbin/rmgroup`, `/usr/bin/mkgroup`. Default for `operatingsystem` == `aix`. Supported features: `manages_aix_lam`, `manages_members`.
>
> **directoryservice**
>
> > Group management using DirectoryService on OS X.
> >
> > Required binaries: `/usr/bin/dscl`. Default for `operatingsystem` == `darwin`. Supported features: `manages_members`.
>
> **groupadd**

Group management via `groupadd` and its ilk. The default for most platforms.

Required binaries: `groupmod`, `groupdel`, `groupadd`. Supported features: `system_groups`.

ldap

Group management via LDAP.

This provider requires that you have valid values for all of the LDAP-related settings in `puppet.conf`, including `ldapbase`. You will almost definitely need settings for `ldapuser` and `ldappassword` in order for your clients to write to LDAP.

Note that this provider will automatically generate a GID for you if you do not specify one, but it is a potentially expensive operation, as it iterates across all existing groups to pick the appropriate next one.

pw

Group management via `pw` on FreeBSD.

Required binaries: `pw`. Default for `operatingsystem` == `freebsd`. Supported features: `manages_members`.

windows_adsi

Local group management for Windows. Nested groups are not supported.

Default for `operatingsystem` == `windows`. Supported features: `manages_members`.

system

Whether the group is a system group with lower GID. Valid values are `true`, `false`.

---

host

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

PARAMETERS

comment

A comment that will be attached to the line with a # character.

ensure

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

host_aliases

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

ip

The host's IP address, IPv4 or IPv6.

name

The host name.

provider

The specific backend to use for this `host` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

parsed

target

The file in which to store service information. Only used by those providers that write to disk. On most systems this defaults to `/etc/hosts`.

---

interface

This represents a router or switch interface. It is possible to manage interface mode (access or trunking, native vlan and encapsulation) and switchport characteristics (speed, duplex).

PARAMETERS

allowed_trunk_vlans

Allowed list of Vlans that this trunk can forward. Valid values are `all`. Values can match `/./`.

description

Interface description.

device_url

The URL at which the router or switch can be reached.

duplex

Interface duplex. Valid values are `auto`, `full`, `half`.

encapsulation

Interface switchport encapsulation. Valid values are `none`, `dot1q`, `isl`.

ensure

The basic property that the resource should be in. Valid values are `present` (also called `no_shutdown`), `absent` (also called `shutdown`).

etherchannel

Channel group this interface is part of. Values can match `/^\d+/`.

ipaddress

IP Address of this interface. Note that it might not be possible to set an interface IP address; it depends on the interface type and device type.

Valid format of ip addresses are:

- IPV4, like 127.0.0.1
- IPV4/prefixlength like 127.0.1.1/24
- IPV6/prefixlength like FE80::21A:2FFF:FE30:ECF0/128
- an optional suffix for IPV6 addresses from this list: `eui-64`, `link-local`

It is also possible to supply an array of values.

mode

Interface switchport mode. Valid values are `access`, `trunk`.

name

The interface's name.

native_vlan

Interface native vlan (for access mode only). Values can match `/^\d+/`.

provider

The specific backend to use for this `interface` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

cisco

Cisco switch/router provider for interface.

speed

Interface speed. Valid values are `auto`. Values can match `/^\d+/`.

---

k5login

Manage the `.k5login` file for a user. Specify the full path to the `.k5login` file as the name, and an

array of principals as the `principals` attribute.

PARAMETERS

ensure

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

mode

> The desired permissions mode of the `.k5login` file. Defaults to `644`.

path

> (Namevar: If omitted, this parameter's value defaults to the resource's title.)
>
> The path to the `.k5login` file to manage. Must be fully qualified.

principals

> The principals present in the `.k5login` file. This should be specified as an array.

provider

> The specific backend to use for this `k5login` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:
>
> **k5login**
>
> > The k5login provider is the only provider for the k5login type.

---

macauthorization

Manage the Mac OS X authorization database. See the Apple developer site for more information.

Note that authorization store directives with hyphens in their names have been renamed to use underscores, as Puppet does not react well to hyphens in identifiers.

Autorequires: If Puppet is managing the `/etc/authorization` file, each macauthorization resource will autorequire it.

PARAMETERS

allow_root

> Corresponds to `allow-root` in the authorization store. 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 in Puppet. Valid values are `user`, `evaluate-mechanisms`, `allow`, `deny`, `rule`.

auth_type

> Type — this can be a `right` or a `rule`. The `comment` type has not yet been implemented. Valid values are `right`, `rule`.

authenticate_user

> Corresponds to `authenticate-user` in the authorization store. 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

> A group which the user must authenticate as a member of. This must be a single group.

k_of_n

> 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 every mechanism — that is, 'n-of-n' — must succeed.

mechanisms

> An array of suitable mechanisms.

name

> The name of the right or rule to be managed. Corresponds to `key` in Authorization Services. The key is the name of a rule. A key uses the same naming conventions as a right. The Security Server uses a rule's key to match the rule with a right. Wildcard keys end with a '.'. The generic rule has an empty key value. Any rights that do not match a specific rule use the generic rule.

provider

> The specific backend to use for this `macauthorization` resource. 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

Whether the session owner automatically matches this rule or right. Corresponds to `session-owner` in the authorization store. Valid values are `true`, `false`.

shared

Whether the Security Server should mark 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 number of seconds in which the credential used by this rule will expire. For maximum security where the user must authenticate every time, set the timeout to 0. For minimum security, remove the timeout attribute so the user authenticates only once per session.

tries

The number of tries allowed.

---

mailalias

Creates an email alias in the local alias database.

PARAMETERS

ensure

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

name

The alias name.

provider

The specific backend to use for this `mailalias` resource. 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 can only create and remove lists; it cannot currently reconfigure them.□

PARAMETERS

admin

> The email address of the administrator.

description

> The description of the mailing list.

ensure

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

mailserver

> The name of the host handling email for the list.

name

> The name of the email list.

password

> The admin password.

provider

> The specific backend to use for this `maillist` resource. 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.

The default provider of this type merely manages the XML plist as reported by the `dscl -mcxexport` command. This is similar to the content property of the file type in Puppet.□

The recommended method of using this type is to use Work Group Manager to manage users and groups on the local computer, record the resulting puppet manifest using the command `puppet resource mcx`, then deploy it to other machines.

Autorequires: If Puppet is managing the user, group, or computer that these MCX settings refer to, the MCX resource will autorequire that user, group, or computer.

FEATURES

- manages_content: The provider can manage MCXSettings as a string.

Provider	manages content
mcxcontent	X

PARAMETERS

content

The XML Plist used as the value of MCXSettings in DirectoryService. This is the standard output from the system command:

```
dscl localhost -mcxexport /Local/Default/<ds_type>/ds_name
```

Note that `ds_type` is capitalized and plural in the dscl command. Requires features manages_content.

ds_name

The name to attach the MCX Setting to. (For example, `localhost` when `ds_type => computer`.) This setting is not required, as it can be automatically discovered when the resource name is parseable. (For example, in `/Groups/admin`, `group` will be used as the dstype.)

ds_type

The DirectoryService type this MCX setting attaches to. Valid values are `user`, `group`, `computer`, `computerlist`.

ensure

Create or remove the MCX setting. Valid values are `present`, `absent`.

name

The name of the resource being managed. The default naming convention follows Directory Service paths:

```
/Computers/localhost
/Groups/admin
/Users/localadmin
```

The `ds_type` and `ds_name` type parameters are not necessary if the default naming convention is followed.

provider

The specific backend to use for this `mcx` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

mcxcontent

MCX Settings management using DirectoryService on OS X.

This provider manages the entire MCXSettings attribute available to some directory services nodes. This management is 'all or nothing' in that discrete application domain key value pairs are not managed by this provider.

It is recommended to use WorkGroup Manager to configure Users, Groups, Computers, or ComputerLists, then use 'ralsh mcx' to generate a puppet manifest from the resulting configuration.

Original Author: Jeff McCune (mccune.jeff@gmail.com)

Required binaries: `/usr/bin/dscl`. Default for `operatingsystem` == `darwin`.
Supported features: `manages_content`.

---

mount

Manages mounted filesystems, including putting mount information into the mount table. The actual behavior depends on the value of the 'ensure' parameter.

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

FEATURES

- refreshable: The provider can remount the filesystem.

Provider	refreshable
parsed	X

PARAMETERS

atboot

Whether to mount the mount at boot. Not all platforms support this.

blockdevice

The device to fsck. This is property is only valid on Solaris, and in most cases will default to the correct value.

device

The device providing the mount. This can be whatever device is supporting by the mount, including network devices or devices specified by UUID rather than device path, depending on the operating system.

dump

Whether to dump the mount. Not all platform support this. Valid values are `1` or `0`. or `2` on FreeBSD, Default is `0`. Values can match `/(0|1)/`, `/(0|1)/`.

ensure

Control what to do with this mount. Set this attribute to `umounted` to make sure the filesystem is in the filesystem table but not mounted (if the filesystem is currently mounted, it will be unmounted). Set it to `absent` to unmount (if necessary) and remove the filesystem from the fstab. Set to `mounted` to add it to the fstab and mount it. Set to `present` to add to fstab but not change mount/unmount status. Valid values are `defined` (also called `present`), `unmounted`, `absent`, `mounted`.

fstype

The mount type. Valid values depend on the operating system. This is a required option.

name

The mount path for the mount.

options

Mount options for the mounts, as they would appear in the fstab.

pass

The pass in which the mount is checked.

path

The deprecated name for the mount point. Please use `name` now.

provider

The specific backend to use for this `mount` resource. You will seldom need to specify this —

Puppet will usually discover the appropriate provider for your platform. Available providers are:

parsed

Required binaries: `mount`, `umount`. Supported features: `refreshable`.

remounts

Whether the mount can be remounted `mount -o remount`. If this is false, then the filesystem will be unmounted and remounted manually, which is prone to failure. Valid values are `true`, `false`.

target

The file in which to store the mount table. Only used by those providers that write to disk.

---

nagios_command

The Nagios type command. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_command.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations. This is an architectural limitation.

PARAMETERS

command_line

Nagios configuration file parameter.

command_name

(Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_command resource.

ensure

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

provider

The specific backend to use for this `nagios_command` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

naginator

target

The target.

use

Nagios configuration file parameter.□

---

nagios_contact

The Nagios type contact. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_contact.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations.□ This is an architectural limitation.

PARAMETERS

address1

Nagios configuration file parameter.□

address2

Nagios configuration file parameter.□

address3

Nagios configuration file parameter.□

address4

Nagios configuration file parameter.□

address5

Nagios configuration file parameter.□

address6

Nagios configuration file parameter.□

alias

Nagios configuration file parameter.□

can_submit_commands

Nagios configuration file parameter.□

contact_name

(Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_contact resource.

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 to use for this `nagios_contact` resource. 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

> The target.

use

> Nagios configuration file parameter.□

---

nagios_contactgroup

The Nagios type contactgroup. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_contactgroup.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations.□ This is an architectural limitation.

PARAMETERS

alias

> Nagios configuration file parameter.□

contactgroup_members

> Nagios configuration file parameter.□

contactgroup_name

> (Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_contactgroup resource.

ensure

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

members

Nagios configuration file parameter.□

provider

The specific backend to use for this `nagios_contactgroup` resource. 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

The target.

use

Nagios configuration file parameter.□

---

nagios_host

The Nagios type host. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_host.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations.□ This is an architectural limitation.

PARAMETERS

action_url

Nagios configuration file parameter.□

active_checks_enabled

Nagios configuration file parameter.□

address

> Nagios configuration file parameter.□

alias

> Nagios configuration file parameter.□

check_command

> Nagios configuration file parameter.□

check_freshness

> Nagios configuration file parameter.□

check_interval

> Nagios configuration file parameter.□

check_period

> Nagios configuration file parameter.□

contact_groups

> Nagios configuration file parameter.□

contacts

> Nagios configuration file parameter.□

display_name

> Nagios configuration file parameter.□

ensure

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

event_handler

> Nagios configuration file parameter.□

event_handler_enabled

> Nagios configuration file parameter.□

failure_prediction_enabled

> Nagios configuration file parameter.□

first_notification_delay□

> Nagios configuration file parameter.□

flap_detection_enabled□

Nagios configuration file parameter.□

flap_detection_options□

Nagios configuration file parameter.□

freshness_threshold

Nagios configuration file parameter.□

high_flap_threshold□

Nagios configuration file parameter.□

host_name

(Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_host resource.

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 to use for this `nagios_host` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

naginator

register

Nagios configuration file parameter.□

retain_nonstatus_information

Nagios configuration file parameter.□

retain_status_information

Nagios configuration file parameter.□

retry_interval

Nagios configuration file parameter.□

stalking_options

Nagios configuration file parameter.□

statusmap_image

Nagios configuration file parameter.□

target

The target.

use

Nagios configuration file parameter.□

vrml_image

Nagios configuration file parameter.□

---

nagios_hostdependency

The Nagios type hostdependency. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_hostdependency.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations.□ This is an architectural limitation.

PARAMETERS

_naginator_name

(Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_hostdependency resource.

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 to use for this `nagios_hostdependency` resource. 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

> The target.

use

> Nagios configuration file parameter.□

---

nagios_hostescalation

The Nagios type hostescalation. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_hostescalation.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations.□ This is an architectural limitation.

PARAMETERS

_naginator_name

> (Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_hostescalation resource.

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 to use for this `nagios_hostescalation` resource. 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

The target.

use

Nagios configuration file parameter.□

---

nagios_hostextinfo

The Nagios type hostextinfo. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_hostextinfo.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations.□ This is an architectural limitation.

PARAMETERS

ensure

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

host_name

(Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_hostextinfo resource.

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 to use for this `nagios_hostextinfo` resource. 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

> The target.

use

> Nagios configuration file parameter.□

vrml_image

> Nagios configuration file parameter.□

---

nagios_hostgroup

The Nagios type hostgroup. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_hostgroup.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations.□ This is an architectural limitation.

PARAMETERS

action_url

> Nagios configuration file parameter.□

alias

> Nagios configuration file parameter.□

ensure

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

hostgroup_members

> Nagios configuration file parameter.□

hostgroup_name

> (Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_hostgroup resource.

members

Nagios configuration file parameter.□

notes

Nagios configuration file parameter.□

notes_url

Nagios configuration file parameter.□

provider

The specific backend to use for this `nagios_hostgroup` resource. 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

The target.

use

Nagios configuration file parameter.□

---

nagios_service

The Nagios type service. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_service.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations.□ This is an architectural limitation.

PARAMETERS

_naginator_name

(Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_service resource.

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 to use for this `nagios_service` resource. 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

The target.

use

Nagios configuration file parameter.□

---

nagios_servicedependency

The Nagios type servicedependency. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_servicedependency.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations.□ This is an architectural limitation.

PARAMETERS

_naginator_name

(Namevar: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_servicedependency resource.

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 to use for this `nagios_servicedependency` resource. 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

The target.

use

Nagios configuration file parameter.□

nagios_serviceescalation

The Nagios type serviceescalation. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_serviceescalation.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations. This is an architectural limitation.

PARAMETERS

_naginator_name

> (Namevar: If omitted, this parameter's value defaults to the resource's title.)
>
> The name of this nagios_serviceescalation resource.

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 to use for this `nagios_serviceescalation` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

naginator

register

Nagios configuration file parameter.□

service_description

Nagios configuration file parameter.□

servicegroup_name

Nagios configuration file parameter.□

target

The 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: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_serviceextinfo resource.

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 to use for this `nagios_serviceextinfo` resource. 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

The target.

use

Nagios configuration file parameter.□

---

nagios_servicegroup

The Nagios type servicegroup. This resource type is autogenerated using the model developed in Naginator, and all of the Nagios types are generated using the same code and the same library.

This type generates Nagios configuration statements in Nagios-parseable configuration files. By default, the statements will be added to `/etc/nagios/nagios_servicegroup.cfg`, but you can send them to a different file by setting their `target` attribute.

You can purge Nagios resources using the `resources` type, but only in the default file locations. This is an architectural limitation.

PARAMETERS

action_url

> Nagios configuration file parameter.

alias

> Nagios configuration file parameter.

ensure

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

members

> Nagios configuration file parameter.

notes

> Nagios configuration file parameter.

notes_url

> Nagios configuration file parameter.

provider

> The specific backend to use for this `nagios_servicegroup` resource. 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: If omitted, this parameter's value defaults to the resource's title.)

The name of this nagios_servicegroup resource.

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 to use for this `nagios_timeperiod` resource. 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

> The target.

thursday

> Nagios configuration file parameter.□

timeperiod_name

> (Namevar: If omitted, this parameter's value defaults to the resource's title.)
>
> The name of this nagios_timeperiod resource.

tuesday

> Nagios configuration file parameter.□

use

> Nagios configuration file parameter.□

wednesday

> Nagios configuration file parameter.□

---

notify

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

PARAMETERS

message

> The message to be sent to the log.

name

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

withpath

> Whether to show the full object path. Defaults to false. Valid values are `true`, `false`.

package

Manage packages. There is a basic dichotomy in package support right now: Some package types (e.g., yum and apt) can retrieve their own package files, while others (e.g., rpm and sun) cannot. For those package formats that cannot retrieve their own files, you can use the `source` parameter to point to the correct file.

Puppet will automatically guess the packaging format that you are using based on the platform you are on, but you can override it using the `provider` parameter; each provider defines what it requires in order to function, and you must meet those requirements to use a given provider.

Autorequires: If Puppet is managing the files specified as a package's `adminfile`, `responsefile`, or `source`, the package resource will autorequire those files.

FEATURES

- holdable: The provider is capable of placing packages on hold such that they are not automatically upgraded as a result of other package dependencies unless explicit action is taken by a user or another package. Held is considered a superset of installed.
- install_options: The provider accepts options to be passed to the installer command.
- installable: The provider can install packages.
- purgeable: The provider can purge packages. This generally means that all traces of the package are removed, including existing configuration files. This feature is thus destructive and should be used with the utmost care.
- uninstallable: The provider can uninstall packages.
- upgradeable: The provider can upgrade to the latest version of a package. This feature is used by specifying `latest` as the desired value for the package.
- versionable: The provider is capable of interrogating the package database for installed version(s), and can select which out of a set of available versions of a package to install if asked.

Provider	holdable	install options	installable	purgeable	uninstallable	upgradeable	versionable
aix			X		X	X	X
appdmg			X				
apple			X				
apt	X		X	X	X	X	X
aptitude	X		X	X	X	X	X
aptrpm			X	X	X	X	X
blastwave			X		X	X	
dpkg	X		X	X	X	X	
fink	X		X	X	X	X	X

freebsd			X		X		
gem			X		X	X	X
hpux			X		X		
macports			X		X	X	X
msi		X	X		X		
nim			X		X	X	X
openbsd			X		X		X
pacman			X		X	X	
pip			X		X	X	X
pkg			X		X	X	
pkgdmg			X				
pkgutil			X		X	X	
portage			X		X	X	X
ports			X		X	X	
portupgrade			X		X	X	
rpm			X		X	X	X
rug			X		X	X	X
sun			X		X	X	
sunfreeware			X		X	X	
up2date			X		X	X	
urpmi			X		X	X	X
yum			X	X	X	X	X
zypper			X		X	X	X

PARAMETERS

adminfile□

> A file containing package defaults for installing packages. This is currently only used on□ Solaris. The value will be validated according to system rules, which in the case of Solaris means that it should either be a fully qualified path or it should be in□ `/var/sadm/install/admin`.

allowcdrom

> Tells apt to allow cdrom sources in the sources.list file. Normally apt will bail if you try this.□ Valid values are `true`, `false`.

category

A read-only parameter set by the package.

configfiles

Whether configfiles should be kept or replaced. Most packages types do not support this parameter. Defaults to `keep`. Valid values are `keep`, `replace`.

description

A read-only parameter set by the package.

ensure

What state the package should be in. On packaging systems that can retrieve new packages on their own, you can choose which package to retrieve by specifying a version number or `latest` as the ensure value. On packaging systems that manage configuration files separately from "normal" system files, you can uninstall config files by specifying `purged` as the ensure value. Valid values are `present` (also called `installed`), `absent`, `purged`, `held`, `latest`. Values can match `/./`.

flavor

Newer versions of OpenBSD support 'flavors', which are further specifications for which type of package you want.

install_options

A hash of additional options to pass when installing a package. These options are package-specific, and should be documented by the software vendor. The most commonly implemented option is `INSTALLDIR`:

```
package { 'mysql':
  ensure          => installed,
  provider        => 'msi',
  source          => 'N:/packages/mysql-5.5.16-winx64.msi',
  install_options => { 'INSTALLDIR' => 'C:\\mysql-5.5' },
}
```

Since these options are passed verbatim to `msiexec`, any file paths specified in `install_options` should use a backslash as the separator character rather than a forward slash. This is the only place in Puppet where backslash separators should be used. Note that backslashes in double-quoted strings must be double-escaped and backslashes in single-quoted strings may be double-escaped. Requires features install_options.

instance

A read-only parameter set by the package.

name

The package name. This is the name that the packaging system uses internally, which is sometimes (especially on Solaris) a name that is basically useless to humans. If you want to abstract package installation, then you can use aliases to provide a common name to packages:

```
# In the 'openssl' class
$ssl = $operatingsystem ? {
  solaris => SMCossl,
  default => openssl
}

# It is not an error to set an alias to the same value as the
# object name.
package { $ssl:
  ensure => installed,
  alias  => openssl
}

. etc. .

$ssh = $operatingsystem ? {
  solaris => SMCossh,
  default => openssh
}

# Use the alias to specify a dependency, rather than
# having another selector to figure it out again.
package { $ssh:
  ensure  => installed,
  alias   => openssh,
  require => Package[openssl]
}
```

platform

A read-only parameter set by the package.

provider

The specific backend to use for this `package` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

aix

Installation from the AIX software directory.

Required binaries: `/usr/sbin/installp`, `/usr/bin/lslpp`. Default for `operatingsystem` == `aix`. Supported features: `installable`, `uninstallable`, `upgradeable`, `versionable`.

appdmg

Package management which copies application bundles to a target.

Required binaries: `/usr/bin/hdiutil`, `/usr/bin/curl`, `/usr/bin/ditto`. Supported features: `installable`.

apple

Package management based on OS X's builtin packaging system. This is essentially the simplest and least functional package system in existence – it only supports installation; no deletion or upgrades. The provider will automatically add the `.pkg` extension, so leave that off when specifying the package name.

Required binaries: `/usr/sbin/installer`. Supported features: `installable`.

apt

Package management via `apt-get`.

Required binaries: `/usr/bin/apt-cache`, `/usr/bin/debconf-set-selections`, `/usr/bin/apt-get`. Default for `operatingsystem` == `debian, ubuntu`. Supported features: `holdable`, `installable`, `purgeable`, `uninstallable`, `upgradeable`, `versionable`.

aptitude

Package management via `aptitude`.

Required binaries: `/usr/bin/apt-cache`, `/usr/bin/aptitude`. Supported features: `holdable`, `installable`, `purgeable`, `uninstallable`, `upgradeable`, `versionable`.

aptrpm

Package management via `apt-get` ported to `rpm`.

Required binaries: `rpm`, `apt-cache`, `apt-get`. Supported features: `installable`, `purgeable`, `uninstallable`, `upgradeable`, `versionable`.

blastwave

Package management using Blastwave.org's `pkg-get` command on Solaris.

Required binaries: `pkg-get`. Supported features: `installable`, `uninstallable`, `upgradeable`.

dpkg

Package management via `dpkg`. Because this only uses `dpkg` and not `apt`, you must specify the source of any packages you want to manage.

Required binaries: `/usr/bin/dpkg-deb`, `/usr/bin/dpkg`, `/usr/bin/dpkg-query`. Supported features: `holdable`, `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: `holdable`, `installable`, `purgeable`, `uninstallable`, `upgradeable`, `versionable`.

freebsd

The specific form of package management on FreeBSD. This is an extremely quirky packaging system, in that it freely mixes between ports and packages. Apparently all of the tools are written in Ruby, so there are plans to rewrite this support to directly use those libraries.

Required binaries: `/usr/sbin/pkg_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`.

macports

Package management using MacPorts on OS X.

Supports MacPorts versions and revisions, but not variants. Variant preferences may be

specified using the MacPorts variants.conf file.

When specifying a version in the Puppet DSL, only specify the version, not the revision. Revisions are only used internally for ensuring the latest version/revision of a port.

Required binaries: `/opt/local/bin/port`. Supported features: `installable`, `uninstallable`, `upgradeable`, `versionable`.

msi

Windows package management by installing and removing MSIs.

This provider requires a `source` attribute, and will accept paths to local files, mapped drives, or UNC paths.

Default for `operatingsystem` == `windows`. Supported features: `install_options`, `installable`, `uninstallable`.

nim

Installation from NIM LPP source.

Required binaries: `/usr/sbin/nimclient`. Supported features: `installable`, `uninstallable`, `upgradeable`, `versionable`.

openbsd

OpenBSD's form of `pkg_add` support.

Required binaries: `pkg_info`, `pkg_add`, `pkg_delete`. Default for `operatingsystem` == `openbsd`. Supported features: `installable`, `uninstallable`, `versionable`.

pacman

Support for the Package Manager Utility (pacman) used in Archlinux.

Required binaries: `/usr/bin/pacman`. Default for `operatingsystem` == `archlinux`. Supported features: `installable`, `uninstallable`, `upgradeable`.

pip

Python packages via `pip`.

Supported features: `installable`, `uninstallable`, `upgradeable`, `versionable`.

pkg

OpenSolaris image packaging system. See pkg(5) for more information

Required binaries: `/usr/bin/pkg`. Supported features: `installable`, `uninstallable`, `upgradeable`.

pkgdmg

Package management based on Apple's Installer.app and DiskUtility.app. This package works by checking the contents of a DMG image for Apple pkg or mpkg files. Any number of pkg or mpkg files may exist in the root directory of the DMG file system. Subdirectories are not checked for packages. See the wiki docs on this provider for more detail.

Required binaries: `/usr/bin/hdiutil`, `/usr/bin/curl`, `/usr/sbin/installer`. Default for `operatingsystem` == `darwin`. Supported features: `installable`.

pkgutil

Package management using Peter Bonivart's `pkgutil` command on Solaris.

Required binaries: `pkgutil`. Supported features: `installable`, `uninstallable`, `upgradeable`.

portage

Provides packaging support for Gentoo's portage system.

Required binaries: `/usr/bin/eix-update`, `/usr/bin/emerge`, `/usr/bin/eix`. Default for `operatingsystem` == `gentoo`. Supported features: `installable`, `uninstallable`, `upgradeable`, `versionable`.

ports

Support for FreeBSD's ports. Note that this, too, mixes packages and ports.

Required binaries: `/usr/local/sbin/portupgrade`, `/usr/local/sbin/portversion`, `/usr/local/sbin/pkg_deinstall`, `/usr/sbin/pkg_info`. Default for `operatingsystem` == `freebsd`. Supported features: `installable`, `uninstallable`, `upgradeable`.

portupgrade

Support for FreeBSD's ports using the portupgrade ports management software. Use the port's full origin as the resource name. eg (ports-mgmt/portupgrade) for the portupgrade port.

Required binaries: `/usr/local/sbin/portupgrade`, `/usr/local/sbin/portversion`, `/usr/local/sbin/portinstall`, `/usr/local/sbin/pkg_deinstall`, `/usr/sbin/pkg_info`. 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: `rpm`, `/usr/bin/rug`. Default for `operatingsystem` == `suse, sles`. 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 `operatingsystem` == `redhat, oel, ovm` and `lsbdistrelease` == `2.1, 3, 4`. Supported features: `installable`, `uninstallable`, `upgradeable`.

urpmi

Support via `urpmi`.

Required binaries: `rpm`, `urpmi`, `urpmq`. Default for `operatingsystem` == `mandriva, mandrake`. Supported features: `installable`, `uninstallable`, `upgradeable`,

`versionable`.

yum

Support via `yum`.

Required binaries: `yum`, `rpm`, `python`. Default for `operatingsystem` == `fedora, centos, redhat`. Supported features: `installable`, `purgeable`, `uninstallable`, `upgradeable`, `versionable`.

zypper

Support for SuSE `zypper` package manager. Found in SLES10sp2+ and SLES11

Required binaries: `/usr/bin/zypper`. Supported features: `installable`, `uninstallable`, `upgradeable`, `versionable`.

responsefile

A file containing any necessary answers to questions asked by the package. This is currently used on Solaris and Debian. The value will be validated according to system rules, but it should generally be a fully qualified path.

root

A read-only parameter set by the package.

source

Where to find the actual package. This must be a local file (or on a network file system) or a URL that your specific packaging type understands; Puppet will not retrieve files for you, although you can manage packages as `file` resources.

status

A read-only parameter set by the package.

type

Deprecated form of `provider`.

vendor

A read-only parameter set by the package.

---

resources

This is a metatype that can manage other resource types. Any metaparams specified here will be passed on to any generated resources, so you can purge umanaged resources but set `noop` to true so the purging is only logged and does not actually happen.

PARAMETERS

name

> The name of the type to be managed.

purge

> Purge unmanaged resources. This will delete any resource that is not specified in your configuration and is not required by any specified resources. Valid values are `true`, `false`.

unless_system_user

> This keeps system users from being purged. By default, it does not purge users whose UIDs are less than or equal to 500, but you can specify a different UID as the inclusive limit. Valid values are `true`, `false`. Values can match `/^\d+$/`.

---

router

Manages connected router.

PARAMETERS

url

> (Namevar: If omitted, this parameter's value defaults to the resource's title.)
>
> An SSH or telnet URL at which to access the router, in the form `ssh://user:pass:enable@host/` or `telnet://user:pass:enable@host/`.

---

schedule

Define schedules for Puppet. Resources can be limited to a schedule by using the `schedule` metaparameter.

Currently, schedules can only be used to stop a resource from being applied; they cannot cause a resource to be applied when it otherwise wouldn't be, and they cannot accurately specify a time when a resource should run.

Every time Puppet applies its configuration, it will apply the set 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 is advisable 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 of the valid periods with the same name as that period (e.g., hourly and daily). Additionally, a schedule named `puppet` is created and used as the default, with the following attributes:

```
schedule { 'puppet':
  period => hourly,
  repeat => 2,
}
```

This will cause resources to be applied every 30 minutes by default.

PARAMETERS

name

The name of the schedule. This name is used to retrieve the schedule when assigning it to an object:

```
schedule { 'daily':
  period => daily,
  range  => "2 - 4",
}

exec { "/usr/bin/apt-get update":
  schedule => 'daily',
}
```

period

The period of repetition for a resource. The default is for a resource to get applied every time Puppet runs.

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 hyphen-separated range within a 24 hour period, and hours must be specified in numbers between 0 and 23, inclusive. Minutes and seconds can optionally 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. Multiple ranges can be applied in array context.

repeat

How often a given resource may be applied in this schedule's `period`. Defaults to 1; must be an integer.

---

scheduled_task

Installs and manages Windows Scheduled Tasks. All attributes except `name`, `command`, and `trigger` are optional; see the description of the `trigger` attribute for details on setting schedules.

PARAMETERS

arguments

Any arguments or flags that should be passed to the command. Multiple arguments should be specified as a space-separated string.

command

The full path to the application to run, without any arguments.

enabled

Whether the triggers for this task should be enabled. This attribute affects every trigger for the task; triggers cannot be enabled or disabled individually. Valid values are `true`, `false`.

ensure

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

name

The name assigned to the scheduled task. This will uniquely identify the task on the system.

password

The password for the user specified in the 'user' attribute. This is only used if specifying a user other than 'SYSTEM'. Since there is no way to retrieve the password used to set the account information for a task, this parameter will not be used to determine if a scheduled task is in sync or not.

provider

The specific backend to use for this `scheduled_task` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

win32_taskscheduler

This provider uses the win32-taskscheduler gem to manage scheduled tasks on Windows.

Puppet requires version 0.2.1 or later of the win32-taskscheduler gem; previous versions can cause "Could not evaluate: The operation completed successfully" errors.

Default for `operatingsystem` == `windows`.

trigger

One or more triggers defining when the task should run. A single trigger is represented as a hash, and multiple triggers can be specified with an array of hashes.

A trigger can contain the following keys:

- For all triggers:
  - `schedule` (Required) — The schedule type. Valid values are `daily`, `weekly`, `monthly`, or `once`.

- `start_time` (Required) — The time of day when the trigger should first become active. Several time formats will work, but we suggest 24-hour time formatted as HH:MM.
- `start_date` — The date when the trigger should first become active. Defaults to "today." Several date formats will work, including special dates like "today," but we suggest formatting dates as YYYY-MM-DD.

- For daily triggers:
  - `every` — How often the task should run, as a number of days. Defaults to 1. ("2" means every other day, "3" means every three days, etc.)

- For weekly triggers:
  - `every` — How often the task should run, as a number of weeks. Defaults to 1. ("2" means every other week, "3" means every three weeks, etc.)
  - `day_of_week` — Which days of the week the task should run, as an array. Defaults to all days. Each day must be one of `mon`, `tues`, `wed`, `thurs`, `fri`, `sat`, `sun`, or `all`.

- For monthly-by-date triggers:
  - `months` — Which months the task should run, as an array. Defaults to all months. Each month must be an integer between 1 and 12.
  - `on` (Required) — Which days of the month the task should run, as an array. Each day must beeither an integer between 1 and 31, or the special value `last,` which is always the last day of the month.

- For monthly-by-weekday triggers:
  - `months` — Which months the task should run, as an array. Defaults to all months. Each month must be an integer between 1 and 12.
  - `day_of_week` (Required) — Which day of the week the task should run, as an array with only one element. Each day must be one of `mon`, `tues`, `wed`, `thurs`, `fri`, `sat`, `sun`, or `all`.
  - `which_occurrence` (Required) — The occurrence of the chosen weekday when the task should run. Must be one of `first`, `second`, `third`, `fourth`, `fifth`, or `last`.

Examples:

```
# Run at 8am on the 1st, 15th, and last day of the month in January,
March,
# May, July, September, and November, starting after August 31st, 2011.
trigger => {
  schedule   => monthly,
  start_date => '2011-08-31',   # Defaults to 'today'
  start_time => '08:00',        # Must be specified
  months     => [1,3,5,7,9,11], # Defaults to all
  on         => [1, 15, last],  # Must be specified
}
```

```
# Run at 8am on the first Monday of the month for January, March, and May,
# starting after August 31st, 2011.
trigger => {
  schedule         => monthly,
  start_date       => '2011-08-31', # Defaults to 'today'
  start_time       => '08:00',      # Must be specified
  months           => [1,3,5],      # Defaults to all
  which_occurrence => first,        # Must be specified
  day_of_week      => [mon],        # Must be specified
}
```

user

The user to run the scheduled task as. Please note that not all security configurations will allow running a scheduled task as 'SYSTEM', and saving the scheduled task under these conditions will fail with a reported error of 'The operation completed successfully'. It is recommended that you either choose another user to run the scheduled task, or alter the security policy to allow v1 scheduled tasks to run as the 'SYSTEM' account. Defaults to 'SYSTEM'.

working_dir

The full path of the directory in which to start the command.

---

selboolean

Manages SELinux booleans on systems with SELinux support. The supported booleans are any of the ones found in `/selinux/booleans/`.

PARAMETERS

name

The name of the SELinux boolean to be managed.

persistent

If set true, SELinux booleans will be written to disk and persist accross reboots. The default is `false`. Valid values are `true`, `false`.

provider

The specific backend to use for this `selboolean` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

getsetsebool

Manage SELinux booleans using the getsebool and setsebool binaries.

Required binaries: `/usr/sbin/getsebool`, `/usr/sbin/setsebool`.

value

> Whether the the SELinux boolean should be enabled or disabled. Valid values are `on`, `off`.

---

selmodule

Manages loading and unloading of SELinux policy modules on the system. Requires SELinux support. See man semodule(8) for more information on SELinux policy modules.

Autorequires: If Puppet is managing the file containing this SELinux policy module (which is either explicitly specified in the `selmodulepath` attribute or will be found at {`selmoduledir`}/{`name`}.pp), the selmodule resource will autorequire that file.

PARAMETERS

ensure

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

name

> The name of the SELinux policy to be managed. You should not include the customary trailing .pp extension.

provider

> The specific backend to use for this `selmodule` resource. 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 the `selmodulepath` attribute is not specified, Puppet will expect to find the module in `<selmoduledir>/<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 `selmoduledir` directory.

syncversion

> If set to `true`, the policy will be reloaded if the version found in the on-disk file differs from the loaded version. If set to `false` (the default) the the only check that will be made is if the policy is loaded at all or not. Valid values are `true`, `false`.

---

service

Manage running services. Service support unfortunately varies widely by platform — some platforms have very little if any concept of a running service, and some have a very codified and powerful concept. Puppet's service support is usually capable of doing the right thing, but the more information you can provide, the better behaviour you will get.

Puppet 2.7 and newer expect init scripts to have a working status command. If this isn't the case for any of your services' init scripts, you will need to set `hasstatus` to false and possibly specify a custom status command in the `status` attribute.

Note that if a `service` receives an event from another resource, the service will get restarted. The actual command to restart the service depends on the platform. You can provide an explicit command for restarting with the `restart` attribute, or you can set `hasrestart` to true to use the init script's restart command; if you do neither, the service's stop and start commands will be used.

FEATURES

- controllable: The provider uses a control variable.
- enableable: The provider can enable and disable the service
- refreshable: The provider can restart the service.

Provider	controllable	enableable	refreshable
base			X
bsd		X	X
daemontools		X	X
debian		X	X
freebsd		X	X
gentoo		X	X
init			X
launchd		X	X
openrc		X	X
redhat		X	X
runit		X	X
service			X

smf		X	X
src			X
systemd		X	X
upstart		X	X
windows		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`, `manual`. 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` command. If this is false and you do not specify a command in the `restart` attribute, the init script's `stop` and `start` commands will be used. Defaults to true; note that this is a change from earlier versions of Puppet. Valid values are `true`, `false`.

hasstatus

> Declare whether the service's init script has a functional status command; defaults to `true`. This attribute's default value changed in Puppet 2.7.0.
>
> The init script's status command must return 0 if the service is running and a nonzero value otherwise. Ideally, these exit codes should conform to the LSB's specification for init script status actions, but Puppet only considers the difference between 0 and nonzero to be relevant.
>
> If a service's init script does not support any kind of status command, you should set

`hasstatus` to false and either provide a specific command using the `status` attribute or expect that Puppet will look for the service name in the process table. Be aware that 'virtual' init scripts (like 'network' under Red Hat systems) will respond poorly to refresh events from other resources if you override the default behavior without providing a status command. Valid values are `true`, `false`.

manifest

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

name

The name of the service to run.

This name is used to find the service; on platforms where services have short system names and long display names, this should be the short name. (To take an example from Windows, you would use "wuauserv" rather than "Automatic Updates.")

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.

Defaults to the name of the service. The pattern can be a simple string or any legal Ruby pattern.

provider

The specific backend to use for this `service` resource. 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 Unix 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. As with `init`-style services, it is preferable to specify start, stop, and status commands.

Required binaries: `kill`. Supported features: `refreshable`.

bsd

FreeBSD's (and probably NetBSD's?) form of `init`-style service management.

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

Supported features: `enableable`, `refreshable`.

daemontools

Daemontools service management.

This provider manages daemons supervised by D.J. Bernstein daemontools. When detecting the service directory it will check, in order of preference:

- `/service`
- `/etc/service`
- `/var/lib/svscan`

The daemon directory should be in one of the following locations:

- `/var/lib/service`
- `/etc`

...or this can be overriden in the resource's attributes:

```
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 shut down the service, not remove the `/path/to/service` link.

Required binaries: `/usr/bin/svc`, `/usr/bin/svstat`. Supported features: `enableable`, `refreshable`.

debian

Debian's form of `init`-style management.

The only differences from `init` are support for enabling and disabling services via `update-rc.d` and the ability to determine enabled status via `invoke-rc.d`.

Required binaries: `/usr/sbin/update-rc.d`, `/usr/sbin/invoke-rc.d`. Default for `operatingsystem` == `debian, ubuntu`. Supported features: `enableable`, `refreshable`.

freebsd

Provider for FreeBSD. Uses the `rcvar` argument of init scripts and parses/edits rc files.

Default for `operatingsystem` == `freebsd`. Supported features: `enableable`, `refreshable`.

gentoo

Gentoo's form of `init`-style service management.

Uses `rc-update` for service enabling and disabling.

Required binaries: `/sbin/rc-update`. Supported features: `enableable`, `refreshable`.

init

Standard `init`-style service management.

Supported features: `refreshable`.

launchd

This provider manages jobs with `launchd`, which is the default service framework for Mac OS X (and may be available for use on other platforms).

For `launchd` documentation, see:

- http://developer.apple.com/macosx/launchd.html
- http://launchd.macosforge.org/

This provider reads plists out of the following directories:

- `/System/Library/LaunchDaemons`
- `/System/Library/LaunchAgents`
- `/Library/LaunchDaemons`

- `/Library/LaunchAgents`

…and builds up a list of services based upon each plist's "Label" entry.

This provider supports:

- ensure => running/stopped,
- enable => true/false
- status
- restart

Here is how the Puppet states correspond to `launchd` states:

- stopped — job unloaded
- started — job loaded
- enabled — 'Disable' removed from job plist file□
- disabled — 'Disable' added to job plist file□

Note that this allows you to do something `launchctl` can't do, which is to be in a state of "stopped/enabled" or "running/disabled".

Note that this provider does not support overriding 'restart' or 'status'.

Required binaries: `/usr/bin/sw_vers`, `/bin/launchctl`, `/usr/bin/plutil`. Default for `operatingsystem` == `darwin`. Supported features: `enableable`, `refreshable`.

openrc

Support for Gentoo's OpenRC initskripts

Uses rc-update, rc-status and rc-service to manage services.

Required binaries: `/bin/rc-status`, `/sbin/rc-update`, `/sbin/rc-service`. Default for `operatingsystem` == `funtoo`. 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` == `redhat, fedora, suse, centos, sles, oel, ovm`. Supported features: `enableable`, `refreshable`.

runit

Runit service management.

This provider manages daemons running supervised by Runit. When detecting the service directory it will check, in order of preference:

- `/service`
- `/var/service`
- `/etc/service`

The daemon directory should be in one of the following locations:

- `/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`.

service

The simplest form of service support.

Supported features: `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/bin/svcs`, `/usr/sbin/svccfg`, `/usr/sbin/svcadm`. Default for

`operatingsystem` == `solaris`. Supported features: `enableable`, `refreshable`.

src

Support for AIX's System Resource controller.

Services are started/stopped based on the `stopsrc` and `startsrc` commands, and some services can be refreshed with `refresh` command.

Enabling and disabling services is not supported, as it requires modifications to `/etc/inittab`. Starting and stopping groups of subsystems is not yet supported.

Required binaries: `/usr/bin/startsrc`, `/usr/bin/lssrc`, `/usr/bin/refresh`, `/usr/bin/stopsrc`. Default for `operatingsystem` == `aix`. Supported features: `refreshable`.

systemd

Manages `systemd` services using `/bin/systemctl`.

Required binaries: `/bin/systemctl`. Supported features: `enableable`, `refreshable`.

upstart

Ubuntu service management with `upstart`.

This provider manages `upstart` jobs, which have replaced `initd` services on Ubuntu. For `upstart` documentation, see http://upstart.ubuntu.com/.

Required binaries: `/sbin/status`, `/sbin/initctl`, `/sbin/start`, `/sbin/restart`, `/sbin/stop`. Default for `operatingsystem` == `ubuntu`. Supported features: `enableable`, `refreshable`.

windows

Support for Windows Service Control Manager (SCM). This provider can start, stop, enable, and disable services, and the SCM provides working status methods for all services.

Control of service groups (dependencies) is not yet supported, nor is running services as a specific user.

Required binaries: `net.exe`. Default for `operatingsystem` == `windows`. 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. This command must return 0 if the service is running and a nonzero value otherwise. Ideally, these exit codes should conform to the LSB's specification for init script status actions, but Puppet only considers the difference between 0 and nonzero to be relevant.

If left unspecified, the status of the service will be determined automatically, usually by looking for the service in the process table.

stop

Specify a stop command manually.

---

ssh_authorized_key

Manages SSH authorized keys. Currently only type 2 keys are supported.

Autorequires: If Puppet is managing the user account in which this SSH key should be installed, the `ssh_authorized_key` resource will autorequire that user.

PARAMETERS

ensure

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

key

The key itself; generally a long string of hex digits.

name

The SSH key comment. This attribute is currently used as a system-wide primary key and therefore has to be unique.

options

Key options, see sshd(8) for possible values. Multiple values should be specified as an array.

provider

The specific backend to use for this `ssh_authorized_key` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

parsed

Parse and generate authorized_keys files for SSH.□

target

The absolute filename in which to store the SSH key. This property is optional and should□ only be used in cases where keys are stored in a non-standard location (i.e. `not in ~user/.ssh/authorized_keys`).

type

The encryption type used: ssh-dss or ssh-rsa. Valid values are `ssh-dss` (also called `dsa`), `ssh-rsa` (also called `rsa`), `ecdsa-sha2-nistp256`, `ecdsa-sha2-nistp384`, `ecdsa-sha2-nistp521`.

user

The user account in which the SSH key should be installed. The resource will automatically depend on this user.

---

sshkey

Installs and manages ssh host keys. At this point, this type only knows how to install keys into `/etc/ssh/ssh_known_hosts`. See the `ssh_authorized_key` type to manage authorized keys.

PARAMETERS

ensure

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

host_aliases

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

key

The key itself; generally a long string of hex digits.

name

The host name that the key is associated with.

provider

The specific backend to use for this `sshkey` resource. 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`), `ecdsa-sha2-nistp256`, `ecdsa-sha2-nistp384`, `ecdsa-sha2-nistp521`.

---

stage

A resource type for specifying run stages. The actual stage should be specified on resources:

```
class { foo: stage => pre }
```

And you must manually control stage order:

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

You automatically get a 'main' stage created, and by default all resources get inserted into that stage.

You can only set stages on class resources, not normal builtin resources.

PARAMETERS

name

> The name of the stage. This will be used as the 'stage' for each resource.

---

tidy

Remove unwanted files based on specific criteria. Multiple criteria are OR'd together, so a file that is too large but is not old enough will still get tidied.

If you don't specify either `age` or `size`, then all files will be removed.

This resource type works by generating a file resource for every file that should be deleted and then letting that resource perform the actual deletion.

PARAMETERS

age

> Tidy files whose age is equal to or greater than the specified time. You can choose seconds, minutes, hours, days, or weeks by specifying the first letter of any of those words (e.g., '1w').

Specifying 0 will remove all files.

backup

Whether tidied files should be backed up. Any values are passed directly to the file resources used for actual file deletion, so consult the `file` type's backup documentation to determine valid values.

matches

One or more (shell type) file glob patterns, which restrict the list of files to be tidied to those whose basenames match at least one of the patterns specified. Multiple patterns can be specified using an array.

Example:

```
tidy { "/tmp":
  age     => "1w",
  recurse => 1,
  matches => [ "[0-9]pub*.tmp", "*.temp", "tmpfile?" ]
}
```

This removes files from `/tmp` if they are one week old or older, are not in a subdirectory and match one of the shell globs given.

Note that the patterns are matched against the basename of each file -- that is, your glob patterns should not have any '/' characters in them, since you are only specifying against the last bit of the file.

Finally, note that you must now specify a non-zero/non-false value for recurse if matches is used, as matches only apply to files found by recursion (there's no reason to use static patterns match against a statically determined path). Requiering explicit recursion clears up a common source of confusion.

path

(Namevar: If omitted, this parameter's value defaults to the resource's title.)

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

recurse

If target is a directory, recursively descend into the directory looking for files to tidy. Valid values are `true`, `false`, `inf`. Values can match `/^[0-9]+$/`.

rmdirs

Tidy directories in addition to files; that is, remove directories whose age is older than the specified criteria. This will only remove empty directories, so all contained files must also be tidied before a directory gets removed. Valid values are `true`, `false`.

size

> Tidy files whose size is equal to or greater than the specified size. Unqualified values are in kilobytes, but b, k, m, g, and t can be appended to specify bytes, kilobytes, megabytes, gigabytes, and terabytes, respectively. Only the first character is significant, so the full word can also be used.

type

> Set the mechanism for determining age. Default: atime. Valid values are `atime`, `mtime`, `ctime`.

---

user

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

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

Autorequires: If Puppet is managing the user's primary group (as provided in the `gid` attribute), the user resource will autorequire that group. If Puppet is managing any role accounts corresponding to the user's roles, the user resource will autorequire those role accounts.

FEATURES

- allows_duplicates: The provider supports duplicate users with the same UID.
- manages_aix_lam: The provider can manage AIX Loadable Authentication Module (LAM) system.
- manages_expiry: The provider can manage the expiry date for a user.
- manages_homedir: The provider can create and remove home directories.
- manages_password_age: The provider can set age requirements and restrictions for passwords.
- manages_passwords: The provider can modify user passwords, by accepting a password hash.
- manages_solaris_rbac: The provider can manage roles and normal users
- system_users: The provider allows you to create system users with lower UIDs.

Provider	allows duplicates	manages aix lam	manages expiry	manages homedir	manages password age	manages passwords	manages solaris rbac	system users
aix		X	X	X	X	X		
directoryservice						X		
hpuxuseradd	X			X				
ldap						X		
pw	X		X	X		X		
user_role_add	X			X	X	X	X	
useradd	X		X	X				X

windows_adsi				X		X		

PARAMETERS

allowdupe

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

attribute_membership

Whether specified attribute value pairs should be treated as the complete list (`inclusive`) or the minimum list (`minimum`) of attribute/value pairs for the user. Defaults to `minimum`. Valid values are `inclusive`, `minimum`.

attributes

Specify AIX attributes for the user in an array of attribute = value pairs. Requires features manages_aix_lam.

auth_membership

Whether specified auths should be considered the complete list (`inclusive`) or the minimum list (`minimum`) of auths the user has. Defaults to `minimum`. 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 the user's full name.

ensure

The basic state that the object should be in. Valid values are `present`, `absent`, `role`.

expiry

The expiry date for this user. Must be provided in a zero-padded YYYY-MM-DD format — e.g. 2010-02-19. Requires features manages_expiry.

gid

The user's primary group. Can be specified numerically or by name.

Note that users on Windows systems do not have a primary group; manage groups with the `groups` attribute instead.

groups

The groups to which the user belongs. The primary group should not be listed, and groups

should be identified by name rather than by GID. Multiple groups should be specified as an array.

home

The home directory of the user. The directory must be created separately and is not currently checked for existence.

ia_load_module

The name of the I&A module to use to manage this user. Requires features manages_aix_lam.

key_membership

Whether specified key/value pairs should be considered the complete list (`inclusive`) or the minimum list (`minimum`) of the user's attributes. Defaults to `minimum`. Valid values are `inclusive`, `minimum`.

keys

Specify user attributes in an array of key = value pairs. Requires features manages_solaris_rbac.

managehome

Whether to manage the home directory when managing the user. Defaults to `false`. Valid values are `true`, `false`.

membership

Whether specified groups should be considered the complete list (`inclusive`) or the minimum list (`minimum`) of groups to which the user belongs. Defaults to `minimum`. Valid values are `inclusive`, `minimum`.

name

The user name. While naming limitations vary by operating system, it is advisable to restrict names to the lowest common denominator, which is a maximum of 8 characters beginning with a letter.

Note that Puppet considers user names to be case-sensitive, regardless of the platform's own rules; be sure to always use the same case when referring to a given user.

password

The user's password, in whatever encrypted format the local system requires.

- Most modern Unix-like systems use salted SHA1 password hashes. You can use Puppet's built-in `sha1` function to generate a hash from a password.
- Mac OS X 10.5 and 10.6 also use salted SHA1 hashes.
- Mac OS X 10.7 (Lion) uses salted SHA512 hashes. The Puppet Labs stdlib module contains

a `str2saltedsha512` function which can generate password hashes for Lion.

- Windows passwords can only be managed in cleartext, as there is no Windows API for setting the password hash.

Be sure to enclose any value that includes a dollar sign ($) in single quotes (') to avoid accidental variable interpolation. Requires features manages_passwords.

password_max_age

The maximum number of days a password may be used before it must be changed. Requires features manages_password_age.

password_min_age

The minimum number of days a password must be used before it may be changed. Requires features manages_password_age.

profile_membership

Whether specified roles should be treated as the complete list (`inclusive`) or the minimum list (`minimum`) of roles of which the user is a member. Defaults to `minimum`. 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 to use for this `user` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:

aix

User management for AIX.

Required binaries: `/usr/sbin/lsgroup`, `/usr/bin/chuser`, `/bin/chpasswd`, `/usr/sbin/lsuser`, `/usr/sbin/rmuser`, `/usr/bin/mkuser`. Default for `operatingsystem` == `aix`. Supported features: `manages_aix_lam`, `manages_expiry`, `manages_homedir`, `manages_password_age`, `manages_passwords`.

directoryservice

User management using DirectoryService on OS X.

Required binaries: `/usr/bin/dscl`. Default for `operatingsystem` == `darwin`. Supported features: `manages_passwords`.

hpuxuseradd

User management for HP-UX. This provider uses the undocumented `-F` switch to HP-UX's special `usermod` binary to work around the fact that its standard `usermod` cannot make changes 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 in `puppet.conf`, including `ldapbase`. You will almost definitely need settings for `ldapuser` and `ldappassword` in order for your clients to 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_expiry`, `manages_homedir`, `manages_passwords`.

user_role_add

User and role management on Solaris, via `useradd` and `roleadd`.

Required binaries: `usermod`, `roleadd`, `roledel`, `rolemod`, `userdel`, `passwd`, `useradd`. Default for `operatingsystem` == `solaris`. Supported features: `allows_duplicates`, `manages_homedir`, `manages_password_age`, `manages_passwords`, `manages_solaris_rbac`.

useradd

User management via `useradd` and its ilk. Note that you will need to install Ruby's

shadow password library (often known as `ruby-libshadow`) if you wish to manage user passwords.

Required binaries: `usermod`, `userdel`, `chage`, `useradd`. Supported features: `allows_duplicates`, `manages_expiry`, `manages_homedir`, `system_users`.

windows_adsi

Local user management for Windows.

Default for `operatingsystem` == `windows`. Supported features: `manages_homedir`, `manages_passwords`.

role_membership

Whether specified roles should be considered the complete list (`inclusive`) or the minimum list (`minimum`) of roles the user has. Defaults to `minimum`. 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.

This attribute cannot be managed on Windows systems.

system

Whether the user is a system user, according to the OS's criteria; on most platforms, a UID less than or equal to 500 indicates a system user. Defaults to `false`. Valid values are `true`, `false`.

uid

The user ID; must be specified numerically. If no user ID is specified when creating a new user, then one will be chosen automatically. This will likely result in the same user having different UIDs on different systems, which is not recommended. This is especially noteworthy when managing the same user on both Darwin and other platforms, since Puppet does UID generation on Darwin, but the underlying tools do so on other platforms.

On Windows, this property is read-only and will return the user's security identifier (SID).

---

vlan

Manages a VLAN on a router or switch.

PARAMETERS

description

> The VLAN's name.

device_url

> The URL of the router or switch maintaining this VLAN.

ensure

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

name

> The numeric VLAN ID. Values can match `/^\d+/`.

provider

> The specific backend to use for this `vlan` resource. You will seldom need to specify this — Puppet will usually discover the appropriate provider for your platform. Available providers are:
>
> ##### cisco
>
> > Cisco switch/router provider for vlans.

---

yumrepo

The client-side description of a yum repository. Repository configurations are found by parsing `/etc/yum.conf` and the files indicated by the `reposdir` option in that file (see `yum.conf(5)` for details).

Most parameters are identical to the ones documented in the `yum.conf(5)` man page.

Continuation lines that yum supports (for the `baseurl`, for example) are not supported. This type does not attempt to read or verify the exinstence of files listed in the `include` attribute.

PARAMETERS

baseurl

> The URL for this repository. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/.*/`.

cost

> Cost of this repository. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/\d+/`.

descr

A human-readable description of the repository. This corresponds to the name parameter in `yum.conf(5)`. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/.*/`.

enabled

Whether this repository is enabled, as represented by a `0` or `1`. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/(0|1)/`.

enablegroups

Whether yum will allow the use of package groups for this repository, as represented by a `0` or `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

The failover methode for this repository; should be 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, as represented by a `0` or `1`. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/(0|1)/`.

gpgkey

The URL for the GPG key with which packages from this repository are signed. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/.*/`.

http_caching

What to cache from this repository. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/packages|all|none/`.

include

The URL of a remote file containing additional yum configuration settings. Puppet does not check for this file's existence or validity. 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 from this repo. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/.*/`.

keepalive

Whether HTTP/1.1 keepalive should be used with this repository, as represented by a `0` or `1`. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/(0|1)/`.

metadata_expire

Number of seconds after which the metadata will expire. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/[0-9]+/`.

mirrorlist

The URL that holds the list of mirrors for this repository. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/.*/`.

name

The name of the repository. This corresponds to the `repositoryid` parameter in `yum.conf(5)`.

priority

Priority of this repository from 1-99. Requires that the `priorities` plugin is installed and enabled. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/[1-9][0-9]?/`.

protect

Enable or disable protection for this repository. Requires that the `protectbase` plugin is installed and enabled. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/(0|1)/`.

proxy

URL to the proxy server for this repository. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/.*/`.

proxy_password

Password for this proxy. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/.*/`.

proxy_username

Username for this proxy. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/.*/`.

timeout

Number of seconds to wait for a connection before timing out. Set this to `absent` to remove it from the file completely. Valid values are `absent`. Values can match `/[0-9]+/`.

---

zfs

Manage zfs. Create destroy and set properties on zfs instances.

Autorequires: If Puppet is managing the zpool at the root of this zfs instance, the zfs resource will autorequire it. If Puppet is managing any parent zfs instances, the zfs resource will autorequire them.

PARAMETERS

aclinherit

The aclinherit property. Valid values are `discard`, `noallow`, `restricted`, `passthrough`, `passthrough-x`.

aclmode

The aclmode property. Valid values are `discard`, `groupmask`, `passthrough`.

atime

The atime property. Valid values are `on`, `off`.

canmount

The canmount property. Valid values are `on`, `off`, `noauto`.

checksum

The checksum property. Valid values are `on`, `off`, `fletcher2`, `fletcher4`, `sha256`.

compression

The compression property. Valid values are `on`, `off`, `lzjb`, `gzip`, `gzip-[1-9]`, `zle`.

copies

The copies property. Valid values are `1`, `2`, `3`.

devices

The devices property. Valid values are `on`, `off`.

ensure

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

exec

The exec property. Valid values are `on`, `off`.

logbias

The logbias property. Valid values are `latency`, `throughput`.

mountpoint

The mountpoint property. Valid values are `<path>`, `legacy`, `none`.

name

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

nbmand

The nbmand property. Valid values are `on`, `off`.

primarycache

The primarycache property. Valid values are `all`, `none`, `metadata`.

provider

The specific backend to use for this `zfs` resource. 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. Valid values are `<size>`, `none`.

readonly

The readonly property. Valid values are `on`, `off`.

recordsize

The recordsize property. Valid values are powers of two between 512 and 128k.

refquota

The refquota property. Valid values are `<size>`, `none`.

refreservation

> The refreservation property. Valid values are `<size>`, `none`.

reservation

> The reservation property. Valid values are `<size>`, `none`.

secondarycache

> The secondarycache property. Valid values are `all`, `none`, `metadata`.

setuid

> The setuid property. Valid values are `on`, `off`.

shareiscsi

> The shareiscsi property. Valid values are `on`, `off`, `type=<type>`.

sharenfs

> The sharenfs property. Valid values are `on`, `off`, share(1M) options

sharesmb

> The sharesmb property. Valid values are `on`, `off`, sharemgr(1M) options

snapdir

> The snapdir property. Valid values are `hidden`, `visible`.

version

> The version property. Valid values are `1`, `2`, `3`, `4`, `current`.

volsize

> The volsize property. Valid values are `<size>`

vscan

> The vscan property. Valid values are `on`, `off`.

xattr

> The xattr property. Valid values are `on`, `off`.

zoned

> The zoned property. Valid values are `on`, `off`.

---

zone

Manages Solaris zones.

Autorequires: If Puppet is managing the directory specified as the root of the zone's filesystem (with the `path` attribute), the zone resource will autorequire that directory.

PARAMETERS

autoboot

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

clone

Instead of installing the zone, clone it from another zone. If the zone root resides on a zfs file system, a snapshot will be used to create the clone; if it resides on a ufs filesystem, a copy of the zone will be used. The zone from which you clone must not be running.

create_args

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

dataset

The list of datasets delegated to the non-global zone from the global zone. All datasets must be zfs filesystem names which are different from the mountpoint.

ensure

The running state of the zone. The valid states directly reflect the states that `zoneadm` provides. The states are linear, in that a zone must be `configured`, then `installed`, and only then can be `running`. Note also that `halt` is currently used to stop zones.

id

The numerical ID of the zone. This number is autogenerated and cannot be changed.

inherit

The list of directories that the zone inherits from the global zone. All directories must be fully qualified.

install_args

Arguments to the `zoneadm` install command. This can be used to create branded zones.

ip

The IP address of the zone. IP addresses must be specified with the interface, separated by a colon, e.g.: bge0:192.168.0.1. For multiple interfaces, specify them in an array.

iptype

The IP stack type of the zone. Valid values are `shared`, `exclusive`.

name

> The name of the zone.

path

> The root of the zone's filesystem. Must be a fully qualified file name. If you include `%s` in the path, then it will be replaced with the zone's name. Currently, you cannot use Puppet to move a zone.

pool

> The resource pool for this zone.

provider

> The specific backend to use for this `zone` resource. 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/zonecfg`, `/usr/sbin/zoneadm`. 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:
>
> ```
> # $confdir/modules/site/templates/sysidcfg.erb
> system_locale=en_US
> timezone=GMT
> terminal=xterms
> security_policy=NONE
> root_password=<%= password %>
> timeserver=localhost
> name_service=DNS {domain_name=<%= domain %> name_server=<%= nameserver %>}
> network_interface=primary {hostname=<%= realhostname %>
>   ip_address=<%= ip %>
>   netmask=<%= netmask %>
>   protocol_ipv6=no
>   default_route=<%= defaultroute %>}
> nfs4_domain=dynamic
> ```

And then call that:

```
zone { myzone:
  ip           => "bge0:192.168.0.23",
  sysidcfg     => template("site/sysidcfg.erb"),
  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 a space separated string.

ensure

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

log

Log disks for this pool. This type does not currently support mirroring of log disks.

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: If omitted, this parameter's value defaults to the resource's title.)

The name for this pool.

provider

The specific backend to use for this `zpool` resource. 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 the `raidz` parameter.

raidz

List of all the devices to raid for this pool. Should be an array of space separated strings:

```
raidz => ["disk1 disk2", "disk3 disk4"],
```

spare

Spare disk(s) for this pool.

This page autogenerated on Tue Jun 18 17:02:09 -0700 2013

Configuration Reference

Configuration Reference

This page is autogenerated; any changes will get overwritten (last generated on Tue Jun 18 17:01:30 -0700 2013)

Configuration Settings

- Each of these settings can be specified in `puppet.conf` or on the command line.
- When using boolean settings on the command line, use `--setting` and `--no-setting` instead of `--setting (true|false)`.
- Settings can be interpolated as `$variables` in other settings; `$environment` is special, in that puppet master will interpolate each agent node's environment instead of its own.
- Multiple values should be specified as comma-separated lists; multiple directories should be separated with the system path separator (usually a colon).
- Settings that take a single file or directory can optionally set the owner, group, and mode for their value: `rundir = $vardir/run { owner = puppet, group = puppet, mode = 644 }`
- The Puppet executables will ignore any setting that isn't relevant to their function.

See the configuration guide for more details.

---

allow_duplicate_certs

Whether to allow a new certificate request to overwrite an existing certificate.

- Default: false

allow_variables_with_dashes

Permit hyphens (`-`) in variable names and issue deprecation warnings about them. This setting should always be `false`; setting it to `true` will cause subtle and wide-ranging bugs. It will be removed in a future version. Hyphenated variables caused major problems in the language, but were allowed between Puppet 2.7.3 and 2.7.14. If you used them during this window, we apologize for the inconvenience — you can temporarily set this to `true` in order to upgrade, and can rename your variables at your leisure. Please revert it to `false` after you have renamed all affected variables.

- Default: false

archive_file_server

During an inspect run, the file bucket server to archive files to if archive_files is set.

- Default: $server

archive_files

During an inspect run, whether to archive files whose contents are audited to a file bucket.

- Default: false

async_storeconfigs

Whether to use a queueing system to provide asynchronous database integration. Requires that `puppetqd` be running and that 'PSON' support for ruby be installed.

- Default: false

authconfig

The configuration file that defines the rights to the different namespaces and methods. This can be used as a coarse-grained authorization system for both `puppet agent` and `puppet master`.

- Default: $confdir/namespaceauth.conf

autoflush

Whether log files should always flush to disk.

- Default: true

autosign

Whether to enable autosign. Valid values are true (which autosigns any key request, and is a very bad idea), false (which never autosigns any key request), and the path to a file, which uses that configuration file to determine which keys to sign.

- Default: $confdir/autosign.conf

bindaddress

The address a listening server should bind to. Mongrel servers default to 127.0.0.1 and WEBrick defaults to 0.0.0.0.

bucketdir

Where FileBucket files are stored.

- Default: $vardir/bucket

ca

Whether the master should function as a certificate authority.

- Default: true

ca_days

How long a certificate should be valid, in days. This setting is deprecated; use `ca_ttl` instead

ca_md

The type of hash used in certificates.□

- Default: md5

ca_name

The name to use the Certificate Authority certificate.□

- Default: Puppet CA: $certname

ca_port

The port to use for the certificate authority.□

- Default: $masterport

ca_server

The server to use for certificate authority requests. It□'s a separate server because it cannot and does not need to horizontally scale.

- Default: $server

ca_ttl

The default TTL for new certificates; valid values must be an integer, optionally followed by one of□ the units 'y' (years of 365 days), 'd' (days), 'h' (hours), or 's' (seconds). The unit defaults to seconds. If this setting is set, ca_days is ignored. Examples are '3600' (one hour) and '1825d', which is the same as '5y' (5 years)

- Default: 5y

cacert

The CA certificate.□

- Default: $cadir/ca_crt.pem

cacrl

The certificate revocation list (CRL) for the CA. Will be used if present but otherwise ignored.□

- Default: $cadir/ca_crl.pem

cadir

The root directory for the certificate authority.□

- Default: $ssldir/ca

cakey

The CA private key.

- Default: $cadir/ca_key.pem

capass

Where the CA stores the password for the private key

- Default: $caprivatedir/ca.pass

caprivatedir

Where the CA stores private certificate information.□

- Default: $cadir/private

capub

The CA public key.

- Default: $cadir/ca_pub.pem

catalog_format

(Deprecated for 'preferred_serialization_format') What format to use to dump the catalog. Only supports 'marshal' and 'yaml'. Only matters on the client, since it asks the server for a specific□ format.

catalog_terminus

Where to get node catalogs. This is useful to change if, for instance, you'd like to pre-compile catalogs and store them in memcached or some other easily-accessed store.

- Default: compiler

cert_inventory

A Complete listing of all certificates□

- Default: $cadir/inventory.txt

certdir

The certificate directory.□

- Default: $ssldir/certs

certdnsnames

The `certdnsnames` setting is no longer functional, after CVE-2011-3872. We ignore the value completely. For your own certificate request you can set `dns_alt_names` in the configuration and it□ will apply locally. There is no configuration option to set DNS alt names, or any other□ `subjectAltName` value, for another nodes certificate. Alternately you can use the `--dns_alt_names`

command line option to set the labels added while generating your own CSR.

certificate_revocation

Whether certificate revocation should be supported by downloading a Certificate Revocation List (CRL) to all clients. If enabled, CA chaining will almost definitely not work.

- Default: true

certname

The name to use when handling certificates. Defaults to the fully qualified domain name.

- Default: (the system's fully qualified domain name)

classfile

The file in which puppet agent stores a list of the classes associated with the retrieved configuration. Can be loaded in the separate `puppet` executable using the `--loadclasses` option.

- Default: $statedir/classes.txt

client_datadir

The directory in which serialized data is stored on the client.

- Default: $vardir/client_data

clientbucketdir

Where FileBucket files are stored locally.

- Default: $vardir/clientbucket

clientyamldir

The directory in which client-side YAML data is stored.

- Default: $vardir/client_yaml

code

Code to parse directly. This is essentially only used by `puppet`, and should only be set if you're writing your own Puppet executable

color

Whether to use colors when logging to the console. Valid values are `ansi` (equivalent to `true`), `html`, and `false`, which produces no color.

- Default: ansi

confdir

The main Puppet configuration directory. The default for this setting is calculated based on the user. If the process is running as root or the user that Puppet is supposed to run as, it defaults to a system directory, but if it's running as any other user, it defaults to being in the user's home directory.

- Default: /etc/puppet

config

The configuration file for doc.

- Default: $confdir/puppet.conf

config_version

How to determine the configuration version. By default, it will be the time that the configuration is parsed, but you can provide a shell script to override how the version is determined. The output of this script will be added to every log message in the reports, allowing you to correlate changes on your hosts to the source version on the server.

configprint

Print the value of a specific configuration setting. If the name of a setting is provided for this, then the value is printed and puppet exits. Comma-separate multiple values. For a list of all values, specify 'all'.

configtimeout

How long the client should wait for the configuration to be retrieved before considering it a failure. This can help reduce flapping if too many clients contact the server at one time.

- Default: 120

couchdb_url

The url where the puppet couchdb database will be created

- Default: http://127.0.0.1:5984/puppet

csrdir

Where the CA stores certificate requests

- Default: $cadir/requests

daemonize

Whether to send the process into the background. This defaults to true on POSIX systems, and to false on Windows (where Puppet currently cannot daemonize).

- Default: true

dbadapter

The type of database to use.

- Default: sqlite3

dbconnections

The number of database connections for networked databases. Will be ignored unless the value is a positive integer.

dblocation

The database cache for client configurations. Used for querying within the language.

- Default: $statedir/clientconfigs.sqlite3

dbmigrate

Whether to automatically migrate the database.

- Default: false

dbname

The name of the database to use.

- Default: puppet

dbpassword

The database password for caching. Only used when networked databases are used.

- Default: puppet

dbport

The database password for caching. Only used when networked databases are used.

dbserver

The database server for caching. Only used when networked databases are used.

- Default: localhost

dbsocket

The database socket location. Only used when networked databases are used. Will be ignored if the value is an empty string.

dbuser

The database user for caching. Only used when networked databases are used.

- Default: puppet

deviceconfig

Path to the device config file for puppet device

- Default: $confdir/device.conf

devicedir

The root directory of devices' $vardir

- Default: $vardir/devices

diff

Which diff command to use when printing differences between files. This setting has no default value on Windows, as standard `diff` is not available, but Puppet can use many third-party diff tools.

- Default: diff

diff_args

Which arguments to pass to the diff command when printing differences between files. The command to use can be chosen with the `diff` setting.

- Default: -u

dns_alt_names

The comma-separated list of alternative DNS names to use for the local host. When the node generates a CSR for itself, these are added to the request as the desired `subjectAltName` in the certificate: additional DNS labels that the certificate is also valid answering as. This is generally required if you use a non-hostname `certname`, or if you want to use `puppet kick` or `puppet resource -H` and the primary certname does not match the DNS name you use to communicate with the host. This is unnecessary for agents, unless you intend to use them as a server for `puppet kick` or remote `puppet resource` management. It is rarely necessary for servers; it is usually helpful only if you need to have a pool of multiple load balanced masters, or for the same master to respond on two physically separate networks under different names.

document_all

Document all resources

- Default: false

downcasefacts

Whether facts should be made all lowercase when sent to the server.

- Default: false

dynamicfacts

Facts that are dynamic; these facts will be ignored when deciding whether changed facts should result in a recompile. Multiple facts should be comma-separated.

- Default: memorysize,memoryfree,swapsize,swapfree

environment

The environment Puppet is running in. For clients (e.g., `puppet agent`) this determines the environment itself, which is used to find modules and much more. For servers (i.e., `puppet master`) this provides the default environment for nodes we know nothing about.

- Default: production

evaltrace

Whether each resource should log when it is being evaluated. This allows you to interactively see exactly what is being done.

- Default: false

external_nodes

An external command that can produce node information. The command's output must be a YAML dump of a hash, and that hash must have a `classes` key and/or a `parameters` key, where `classes` is an array or hash and `parameters` is a hash. For unknown nodes, the command should exit with a non-zero exit code. This command makes it straightforward to store your node mapping information in other data sources like databases.

- Default: none

factdest

Where Puppet should store facts that it pulls down from the central server.

- Default: $vardir/facts/

factpath

Where Puppet should look for facts. Multiple directories should be separated by the system path separator character. (The POSIX path separator is ':', and the Windows path separator is ';'.)

- Default: $vardir/lib/facter:$vardir/facts

facts_terminus

The node facts terminus.

- Default: facter

factsignore

What files to ignore when pulling down facts.

- Default: .svn CVS

factsource

From where to retrieve facts. The standard Puppet `file` type is used for retrieval, so anything that is a valid file source can be used here.

- Default: puppet://$server/facts/

factsync

Whether facts should be synced with the central server.

- Default: false

fileserverconfig

Where the fileserver configuration is stored.

- Default: $confdir/fileserver.conf

filetimeout

The minimum time to wait (in seconds) between checking for updates in configuration files. This timeout determines how quickly Puppet checks whether a file (such as manifests or templates) has changed on disk.

- Default: 15

freeze_main

Freezes the 'main' class, disallowing any code to be added to it. This essentially means that you can't have any code outside of a node, class, or definition other than in the site manifest.

- Default: false

genconfig

Whether to just print a configuration to stdout and exit. Only makes sense when used interactively. Takes into account arguments specified on the CLI.

- Default: false

genmanifest

Whether to just print a manifest to stdout and exit. Only makes sense when used interactively. Takes into account arguments specified on the CLI.

- Default: false

graph

Whether to create dot graph files for the different configuration graphs. These dot files can be interpreted by tools like OmniGraffle or dot (which is part of ImageMagick).

- Default: false

graphdir

Where to store dot-outputted graphs.

- Default: $statedir/graphs

group

The group puppet master should run as.

- Default: puppet

hostcert

Where individual hosts store and look for their certificates.

- Default: $certdir/$certname.pem

hostcrl

Where the host's certificate revocation list can be found. This is distinct from the certificate authority's CRL.

- Default: $ssldir/crl.pem

hostcsr

Where individual hosts store and look for their certificate requests.

- Default: $ssldir/csr_$certname.pem

hostprivkey

Where individual hosts store and look for their private key.

- Default: $privatekeydir/$certname.pem

hostpubkey

Where individual hosts store and look for their public key.

- Default: $publickeydir/$certname.pem

http_compression

Allow http compression in REST communication with the master. This setting might improve performance for agent -> master communications over slow WANs. Your puppet master needs to support compression (usually by activating some settings in a reverse-proxy in front of the puppet master, which rules out webrick). It is harmless to activate this settings if your master doesn't

support compression, but if it supports it, this setting might reduce performance on high-speed LANs.

- Default: false

http_proxy_host

The HTTP proxy host to use for outgoing connections. Note: You may need to use a FQDN for the server hostname when using a proxy.

- Default: none

http_proxy_port

The HTTP proxy port to use for outgoing connections

- Default: 3128

httplog

Where the puppet agent web server logs.

- Default: $logdir/http.log

ignorecache

Ignore cache and always recompile the configuration. This is useful for testing new configurations, where the local cache may in fact be stale even if the timestamps are up to date - if the facts change or if the server changes.

- Default: false

ignoreimport

If true, allows the parser to continue without requiring all files referenced with `import` statements to exist. This setting was primarily designed for use with commit hooks for parse-checking.

- Default: false

ignoreschedules

Boolean; whether puppet agent should ignore schedules. This is useful for initial puppet agent runs.

- Default: false

inventory_port

The port to communicate with the inventory_server.

- Default: $masterport

inventory_server

The server to send facts to.

- Default: $server

inventory_terminus

Should usually be the same as the facts terminus

- Default: $facts_terminus

keylength

The bit length of keys.

- Default: 4096

lastrunfile

Where puppet agent stores the last run report summary in yaml format.

- Default: $statedir/last_run_summary.yaml

lastrunreport

Where puppet agent stores the last run report in yaml format.

- Default: $statedir/last_run_report.yaml

ldapattrs

The LDAP attributes to include when querying LDAP for nodes. All returned attributes are set as variables in the top-level scope. Multiple values should be comma-separated. The value 'all' returns all attributes.

- Default: all

ldapbase

The search base for LDAP searches. It's impossible to provide a meaningful default here, although the LDAP libraries might have one already set. Generally, it should be the 'ou=Hosts' branch under your main directory.

ldapclassattrs

The LDAP attributes to use to define Puppet classes. Values should be comma-separated.

- Default: puppetclass

ldapnodes

Whether to search for node configurations in LDAP. See http://projects.puppetlabs.com/projects/puppet/wiki/LDAP_Nodes for more information.

- Default: false

ldapparentattr

The attribute to use to define the parent node.□

- Default: parentnode

ldappassword

The password to use to connect to LDAP.

ldapport

The LDAP port. Only used if `ldapnodes` is enabled.

- Default: 389

ldapserver

The LDAP server. Only used if `ldapnodes` is enabled.

- Default: ldap

ldapssl

Whether SSL should be used when searching for nodes. Defaults to false because SSL usually requires certificates to be set up on the client side.□

- Default: false

ldapstackedattrs

The LDAP attributes that should be stacked to arrays by adding the values in all hierarchy elements of the tree. Values should be comma-separated.

- Default: puppetvar

ldapstring

The search string used to find an LDAP node.□

- Default: (&(objectclass=puppetClient)(cn=%s))

ldaptls

Whether TLS should be used when searching for nodes. Defaults to false because TLS usually requires certificates to be set up on the client side.□

- Default: false

ldapuser

The user to use to connect to LDAP. Must be specified as a full DN.□

lexical

Whether to use lexical scoping (vs. dynamic).

- Default: false

libdir

An extra search path for Puppet. This is only useful for those files that Puppet will load on demand, and is only guaranteed to work for those cases. In fact, the autoload mechanism is responsible for making sure this directory is in Ruby's search path

- Default: $vardir/lib

listen

Whether puppet agent should listen for connections. If this is true, then puppet agent will accept incoming REST API requests, subject to the default ACLs and the ACLs set in the `rest_authconfig` file. Puppet agent can respond usefully to requests on the `run`, `facts`, `certificate`, and `resource` endpoints.

- Default: false

localcacert

Where each client stores the CA certificate.

- Default: $certdir/ca.pem

localconfig

Where puppet agent caches the local configuration. An extension indicating the cache format is added automatically.

- Default: $statedir/localconfig

logdir

The Puppet log directory.

- Default: $vardir/log

manage_internal_file_permissions

Whether Puppet should manage the owner, group, and mode of files it uses internally

- Default: true

manifest

The entry-point manifest for puppet master.

- Default: $manifestdir/site.pp

manifestdir

Where puppet master looks for its manifests.

- Default: $confdir/manifests

masterhttplog

Where the puppet master web server logs.

- Default: $logdir/masterhttp.log

masterlog

Where puppet master logs. This is generally not used, since syslog is the default log destination.

- Default: $logdir/puppetmaster.log

masterport

Which port puppet master listens on.

- Default: 8140

maximum_uid

The maximum allowed UID. Some platforms use negative UIDs but then ship with tools that do not know how to handle signed ints, so the UIDs show up as huge numbers that can then not be fed back into the system. This is a hackish way to fail in a slightly more useful way when that happens.

- Default: 4294967290

mkusers

Whether to create the necessary user and group that puppet agent will run as.

- Default: false

module_repository

The module repository

- Default: http://forge.puppetlabs.com

module_working_dir

The directory into which module tool data is stored

- Default: $vardir/puppet-module

modulepath

The search path for modules, as a list of directories separated by the system path separator character. (The POSIX path separator is ‘:’, and the Windows path separator is ‘;’.)

- Default: $confdir/modules:/usr/share/puppet/modules

name

The name of the application, if we are running as one. The default is essentially $0 without the path or `.rb`.

- Default: doc

node_name

How the puppet master determines the client's identity and sets the 'hostname', 'fqdn' and 'domain' facts for use in the manifest, in particular for determining which 'node' statement applies to the client. Possible values are 'cert' (use the subject's CN in the client's certificate) and 'facter' (use the hostname that the client reported in its facts)

- Default: cert

node_name_fact

The fact name used to determine the node name used for all requests the agent makes to the master. WARNING: This setting is mutually exclusive with node_name_value. Changing this setting also requires changes to the default auth.conf configuration on the Puppet Master. Please see http://links.puppetlabs.com/node_name_fact for more information.

node_name_value

The explicit value used for the node name for all requests the agent makes to the master. WARNING: This setting is mutually exclusive with node_name_fact. Changing this setting also requires changes to the default auth.conf configuration on the Puppet Master. Please see http://links.puppetlabs.com/node_name_value for more information.

- Default: $certname

node_terminus

Where to find information about nodes.

- Default: plain

noop

Whether puppet agent should be run in noop mode.

- Default: false

onetime

Run the configuration once, rather than as a long-running daemon. This is useful for interactively running puppetd.

- Default: false

passfile

Where puppet agent stores the password for its private key. Generally unused.

- Default: $privatedir/password

path

The shell search path. Defaults to whatever is inherited from the parent process.

- Default: none

pidfile

The pid file

- Default: $rundir/$name.pid

plugindest

Where Puppet should store plugins that it pulls down from the central server.

- Default: $libdir

pluginsignore

What files to ignore when pulling down plugins.

- Default: .svn CVS .git

pluginsource

From where to retrieve plugins. The standard Puppet `file` type is used for retrieval, so anything that is a valid file source can be used here.

- Default: puppet://$server/plugins

pluginsync

Whether plugins should be synced with the central server.

- Default: false

postrun_command

A command to run after every agent run. If this command returns a non-zero return code, the entire Puppet run will be considered to have failed, even though it might have performed work during the normal run.

preferred_serialization_format

The preferred means of serializing ruby instances for passing over the wire. This won't guarantee that all instances will be serialized using this method, since not all classes can be guaranteed to support this format, but it will be used for all classes that support it.

- Default: pson

prerun_command

A command to run before every agent run. If this command returns a non-zero return code, the entire Puppet run will fail.

privatedir

Where the client stores private certificate information.

- Default: $ssldir/private

privatekeydir

The private key directory.

- Default: $ssldir/private_keys

publickeydir

The public key directory.

- Default: $ssldir/public_keys

puppetdlockfile

A lock file to temporarily stop puppet agent from doing anything.

- Default: $statedir/puppetdlock

puppetdlog

The log file for puppet agent. This is generally not used.

- Default: $logdir/puppetd.log

puppetport

Which port puppet agent listens on.

- Default: 8139

queue_source

Which type of queue to use for asynchronous processing. If your stomp server requires authentication, you can include it in the URI as long as your stomp client library is at least 1.1.1

- Default: stomp://localhost:61613/

queue_type

Which type of queue to use for asynchronous processing.

- Default: stomp

rails_loglevel

The log level for Rails connections. The value must be a valid log level within Rails. Production environments normally use `info` and other environments normally use `debug`.

- Default: info

railslog

Where Rails-specific logs are sent□

- Default: $logdir/rails.log

report

Whether to send reports after every transaction.

- Default: true

report_port

The port to communicate with the report_server.

- Default: $masterport

report_server

The server to send transaction reports to.

- Default: $server

reportdir

The directory in which to store reports received from the client. Each client gets a separate subdirectory.

- Default: $vardir/reports

reportfrom

The 'from' email address for the reports.

- Default: report@(the system's fully qualified domain name)□

reports

The list of reports to generate. All reports are looked for in `puppet/reports/name.rb`, and multiple report names should be comma-separated (whitespace is okay).

- Default: store

reportserver

(Deprecated for 'report_server') The server to which to send transaction reports.

- Default: $server

reporturl

The URL used by the http reports processor to send reports

- Default: http://localhost:3000/reports/upload

req_bits

The bit length of the certificates.

- Default: 4096

requestdir

Where host certificate requests are stored.

- Default: $ssldir/certificate_requests

resourcefile

The file in which puppet agent stores a list of the resources associated with the retrieved configuration.

- Default: $statedir/resources.txt

rest_authconfig

The configuration file that defines the rights to the different rest indirections. This can be used as a fine-grained authorization system for `puppet master`.

- Default: $confdir/auth.conf

route_file

The YAML file containing indirector route configuration.

- Default: $confdir/routes.yaml

rrddir

The directory where RRD database files are stored. Directories for each reporting host will be created under this directory.

- Default: $vardir/rrd

rrdinterval

How often RRD should expect data. This should match how often the hosts report back to the server.

- Default: $runinterval

run_mode

The effective ‘run mode’ of the application: master, agent, or user.□

- Default: master

rundir

Where Puppet PID files are kept.□

- Default: $vardir/run

runinterval

How often puppet agent applies the client configuration; in seconds. Note that a runinterval of 0□ means “run continuously” rather than “never run.” If you want puppet agent to never run, you should start it with the `--no-client` option.

- Default: 1800

sendmail

Where to find the sendmail binary with which to send email.□

- Default: /usr/sbin/sendmail

serial

Where the serial number for certificates is stored.□

- Default: $cadir/serial

server

The server to which server puppet agent should connect

- Default: puppet

server_datadir

The directory in which serialized data is stored, usually in a subdirectory.

- Default: $vardir/server_data

servertype

The type of server to use. Currently supported options are webrick and mongrel. If you use mongrel, you will need a proxy in front of the process or processes, since Mongrel cannot speak SSL.

- Default: webrick

show_diff□

Whether to log and report a contextual diff when files are being replaced. This causes partial file□ contents to pass through Puppet’s normal logging and reporting system, so this setting should be

used with caution if you are sending Puppet's reports to an insecure destination. This feature currently requires the `diff/lcs` Ruby library.

- Default: false

signeddir

Where the CA stores signed certificates.

- Default: $cadir/signed

smtpserver

The server through which to send email reports.

- Default: none

splay

Whether to sleep for a pseudo-random (but consistent) amount of time before a run.

- Default: false

splaylimit

The maximum time to delay before runs. Defaults to being the same as the run interval.

- Default: $runinterval

ssl_client_header

The header containing an authenticated client's SSL DN. Only used with Mongrel. This header must be set by the proxy to the authenticated client's SSL DN (e.g., `/CN=puppet.puppetlabs.com`). See http://projects.puppetlabs.com/projects/puppet/wiki/Using_Mongrel for more information.

- Default: HTTP_X_CLIENT_DN

ssl_client_verify_header

The header containing the status message of the client verification. Only used with Mongrel. This header must be set by the proxy to 'SUCCESS' if the client successfully authenticated, and anything else otherwise. See http://projects.puppetlabs.com/projects/puppet/wiki/Using_Mongrel for more information.

- Default: HTTP_X_CLIENT_VERIFY

ssldir

Where SSL certificates are kept.

- Default: $confdir/ssl

statedir

The directory where Puppet state is stored. Generally, this directory can be removed without causing harm (although it might result in spurious service restarts).

- Default: $vardir/state

statefile

Where puppet agent and puppet master store state associated with the running configuration. In the case of puppet master, this file reflects the state discovered through interacting with clients.

- Default: $statedir/state.yaml

storeconfigs

Whether to store each client's configuration, including catalogs, facts, and related data. This also enables the import and export of resources in the Puppet language - a mechanism for exchange resources between nodes. By default this uses ActiveRecord and an SQL database to store and query the data; this, in turn, will depend on Rails being available. You can adjust the backend using the storeconfigs_backend setting.

- Default: false

storeconfigs_backend

Configure the backend terminus used for StoreConfigs. By default, this uses the ActiveRecord store, which directly talks to the database from within the Puppet Master process.

- Default: active_record

strict_hostname_checking

Whether to only search for the complete hostname as it is in the certificate when searching for node information in the catalogs.

- Default: false

summarize

Whether to print a transaction summary.

- Default: false

syslogfacility

What syslog facility to use when logging to syslog. Syslog has a fixed list of valid facilities, and you must choose one of those; you cannot just make one up.

- Default: daemon

tagmap

The mapping between reporting tags and email addresses.

- Default: $confdir/tagmail.conf

tags

Tags to use to find resources. If this is set, then only resources tagged with the specified tags will be applied. Values must be comma-separated.

templatedir

Where Puppet looks for template files. Can be a list of colon-separated directories.

- Default: $vardir/templates

thin_storeconfigs

Boolean; whether Puppet should store only facts and exported resources in the storeconfigs database. This will improve the performance of exported resources with the older `active_record` backend, but will disable external tools that search the storeconfigs database. Thinning catalogs is generally unnecessary when using PuppetDB to store catalogs.

- Default: false

trace

Whether to print stack traces on some errors

- Default: false

use_cached_catalog

Whether to only use the cached catalog rather than compiling a new catalog on every run. Puppet can be run with this enabled by default and then selectively disabled when a recompile is desired.

- Default: false

usecacheonfailure

Whether to use the cached configuration when the remote configuration will not compile. This option is useful for testing new configurations, where you want to fix the broken configuration rather than reverting to a known-good one.

- Default: true

user

The user puppet master should run as.

- Default: puppet

vardir

Where Puppet stores dynamic and growing data. The default for this setting is calculated specially, like `confdir`_.

- Default: /var/lib/puppet

yamldir

The directory in which YAML data is stored, usually in a subdirectory.

- Default: $vardir/yaml

zlib

Boolean; whether to use the zlib library

- Default: true

This page autogenerated on Tue Jun 18 17:01:30 -0700 2013

Function Reference

Function Reference

This page is autogenerated; any changes will get overwritten (last generated on Tue Jun 18 17:01:37 -0700 2013)

There are two types of functions in Puppet: Statements and rvalues. Statements stand on their own and do not return arguments; they are used for performing stand-alone work like importing. Rvalues return values and can only be used in a statement requiring a value, such as an assignment or a case statement.

Functions execute on the Puppet master. They do not execute on the Puppet agent.
Hence they only have access to the commands and data available on the Puppet master host.

Here are the functions available in Puppet:

alert

Log a message on the server at level alert.

- Type: statement

create_resources

Converts a hash into a set of resources and adds them to the catalog.

This function takes two mandatory arguments: a resource type, and a hash describing a set of resources. The hash should be in the form `{title => {parameters} }`:

```
# A hash of user resources:
$myusers = {
  'nick' => { uid    => '1330',
              group  => allstaff,
              groups => ['developers', 'operations', 'release'], }
  'dan'  => { uid    => '1308',
              group  => allstaff,
              groups => ['developers', 'prosvc', 'release'], }
}

create_resources(user, $myusers)
```

A third, optional parameter may be given, also as a hash:

```
$defaults = {
  'ensure'   => present,
  'provider' => 'ldap',
}
```

```
create_resources(user, $myusers, $defaults)
```

The values given on the third argument are added to the parameters of each resource present in the set given on the second argument. If a parameter is present on both the second and third arguments, the one on the second argument takes precedence.

This function can be used to create defined resources and classes, as well as native resources.

- Type: statement

crit

Log a message on the server at level crit.

- Type: statement

debug

Log a message on the server at level debug.

- Type: statement

defined

Determine whether a given class or resource type is defined. This function can also determine whether a specific resource has been declared. Returns true or false. Accepts class names, type names, and resource references.

The `defined` function checks both native and defined types, including types provided as plugins via modules. Types and classes are both checked using their names:

```
defined("file")
defined("customtype")
defined("foo")
defined("foo::bar")
```

Resource declarations are checked using resource references, e.g. `defined( File['/tmp/myfile'] )`. Checking whether a given resource has been declared is, unfortunately, dependent on the parse order of the configuration, and the following code will not work:

```
if defined(File['/tmp/foo']) {
    notify("This configuration includes the /tmp/foo file.")
}
file {"/tmp/foo":
    ensure => present,
}
```

However, this order requirement refers to parse order only, and ordering of resources in the configuration graph (e.g. with `before` or `require`) does not affect the behavior of `defined`.

- Type: rvalue

emerg

Log a message on the server at level emerg.

- Type: statement

err

Log a message on the server at level err.

- Type: statement

extlookup

This is a parser function to read data from external files, this version uses CSV files but the concept can easily be adjust for databases, yaml or any other queryable data source.

The object of this is to make it obvious when it's being used, rather than magically loading data in when an module is loaded I prefer to look at the code and see statements like:

```
$snmp_contact = extlookup("snmp_contact")
```

The above snippet will load the snmp_contact value from CSV files, this in its own is useful but a common construct in puppet manifests is something like this:

```
case $domain {
  "myclient.com": { $snmp_contact = "John Doe <john@myclient.com>" }
  default:        { $snmp_contact = "My Support <support@my.com>" }
}
```

Over time there will be a lot of this kind of thing spread all over your manifests and adding an additional client involves grepping through manifests to find all the places where you have constructs like this.

This is a data problem and shouldn't be handled in code, a using this function you can do just that.

First you configure it in site.pp:

```
$extlookup_datadir = "/etc/puppet/manifests/extdata"
$extlookup_precedence = ["%{fqdn}", "domain_%{domain}", "common"]
```

The array tells the code how to resolve values, first it will try to find it in web1.myclient.com.csv then in domain_myclient.com.csv and finally in common.csv□

Now create the following data files in /etc/puppet/manifests/extdata:□

```
domain_myclient.com.csv:
  snmp_contact,John Doe <john@myclient.com>
  root_contact,support@%{domain}
  client_trusted_ips,192.168.1.130,192.168.10.0/24

common.csv:
  snmp_contact,My Support <support@my.com>
  root_contact,support@my.com
```

Now you can replace the case statement with the simple single line to achieve the exact same outcome:

$snmp_contact = extlookup(“snmp_contact”)

The above code shows some other features, you can use any fact or variable that is in scope by simply using %{varname} in your data files, you can return arrays by just having multiple values in□ the csv after the initial variable name.

In the event that a variable is nowhere to be found a critical error will be raised that will prevent your manifest from compiling, this is to avoid accidentally putting in empty values etc. You can however specify a default value:

$ntp_servers = extlookup(“ntp_servers”, “1.${country}.pool.ntp.org”)

In this case it will default to “1.${country}.pool.ntp.org” if nothing is defined in any data file.□

You can also specify an additional data file to search first before any others at use time, for□ example:

```
$version = extlookup("rsyslog_version", "present", "packages")
package{"rsyslog": ensure => $version }
```

This will look for a version configured in packages.csv and then in the rest as configured by□ $extlookup_precedence if it’s not found anywhere it will default to `present`, this kind of use case makes puppet a lot nicer for managing large amounts of packages since you do not need to edit a load of manifests to do simple things like adjust a desired version number.

Precedence values can have variables embedded in them in the form %{fqdn}, you could for example do:

```
$extlookup_precedence = ["hosts/%{fqdn}", "common"]
```

This will result in /path/to/extdata/hosts/your.box.com.csv being searched.

This is for back compatibility to interpolate variables with %. % interpolation is a workaround for a problem that has been fixed: Puppet variable interpolation at top scope used to only happen on each run.

- Type: rvalue

fail

Fail with a parse error.

- Type: statement

file

Return the contents of a file. Multiple files can be passed, and the first file that exists will be read in.

- Type: rvalue

fqdn_rand

Generates random numbers based on the node's fqdn. Generated random values will be a range from 0 up to and excluding n, where n is the first parameter. The second argument specifies a number to add to the seed and is optional, for example:

```
$random_number = fqdn_rand(30)
$random_number_seed = fqdn_rand(30,30)
```

- Type: rvalue

generate

Calls an external command on the Puppet master and returns the results of the command. Any arguments are passed to the external command as arguments. If the generator does not exit with return code of 0, the generator is considered to have failed and a parse error is thrown. Generators can only have file separators, alphanumerics, dashes, and periods in them. This function will attempt to protect you from malicious generator calls (e.g., those with '..' in them), but it can never be entirely safe. No subshell is used to execute generators, so all shell metacharacters are passed directly to the generator.

- Type: rvalue

include

Evaluate one or more classes.

- Type: statement

info

Log a message on the server at level info.

- Type: statement

inline_template

Evaluate a template string and return its value. See the templating docs for more information. Note that if multiple template strings are specified, their output is all concatenated and returned as the output of the function.

- Type: rvalue

md5

Returns a MD5 hash value from a provided string.

- Type: rvalue

notice

Log a message on the server at level notice.

- Type: statement

realize

Make a virtual object real. This is useful when you want to know the name of the virtual object and don't want to bother with a full collection. It is slightly faster than a collection, and, of course, is a bit shorter. You must pass the object using a reference; e.g.: `realize User[luke]`.

- Type: statement

regsubst

Perform regexp replacement on a string or array of strings.

- Parameters (in order):
  - target The string or array of strings to operate on. If an array, the replacement will be performed on each of the elements in the array, and the return value will be an array.
  - regexp The regular expression matching the target string. If you want it anchored at the start and or end of the string, you must do that with ^ and $ yourself.
  - replacement Replacement string. Can contain backreferences to what was matched using \0 (whole match), \1 (first set of parentheses), and so on.
  - flags Optional. String of single letter flags for how the regexp is interpreted:

        - E Extended regexps
        - I Ignore case in regexps
        - M Multiline regexps
        - G Global replacement; all occurrences of the regexp in each target string will be replaced. Without this, only the first occurrence will be replaced.□
    - encoding Optional. How to handle multibyte characters. A single-character string with the following values:
        - N None
        - E EUC
        - S SJIS
        - U UTF-8

- Examples

Get the third octet from the node's IP address:

```
$i3 = regsubst($ipaddress,'^(\d+)\.(\d+)\.(\d+)\.(\d+)$','\3')
```

Put angle brackets around each octet in the node's IP address:

```
$x = regsubst($ipaddress, '([0-9]+)', '<\1>', 'G')
```

- Type: rvalue

require

Evaluate one or more classes, adding the required class as a dependency.

The relationship metaparameters work well for specifying relationships between individual resources, but they can be clumsy for specifying relationships between classes. This function is a superset of the 'include' function, adding a class relationship so that the requiring class depends on the required class.

Warning: using require in place of include can lead to unwanted dependency cycles.

For instance the following manifest, with 'require' instead of 'include' would produce a nasty dependence cycle, because notify imposes a before between File[/foo] and Service[foo]:

```
class myservice {
  service { foo: ensure => running }
}

class otherstuff {
```

```
  include myservice
  file { '/foo': notify => Service[foo] }
}
```

Note that this function only works with clients 0.25 and later, and it will fail if used with earlier clients.

- Type: statement

search

Add another namespace for this class to search. This allows you to create classes with sets of definitions and add those classes to another class's search path.□

- Type: statement

sha1

Returns a SHA1 hash value from a provided string.

- Type: rvalue

shellquote

Quote and concatenate arguments for use in Bourne shell.

Each argument is quoted separately, and then all are concatenated with spaces. If an argument is an array, the elements of that array is interpolated within the rest of the arguments; this makes it possible to have an array of arguments and pass that array to shellquote instead of having to specify each argument individually in the call.

- Type: rvalue

split

Split a string variable into an array using the specified split regexp.□

Example:

```
$string     = 'v1.v2:v3.v4'
$array_var1 = split($string, ':')
$array_var2 = split($string, '[.]')
$array_var3 = split($string, '[.:]')
```

`$array_var1` now holds the result `['v1.v2', 'v3.v4']`, while `$array_var2` holds `['v1', 'v2:v3', 'v4']`, and `$array_var3` holds `['v1', 'v2', 'v3', 'v4']`.

Note that in the second example, we split on a literal string that contains a regexp meta-character

(.), which must be escaped. A simple way to do that for a single character is to enclose it in square brackets; a backslash will also escape a single character.

- Type: rvalue

sprintf

Perform printf-style formatting of text.

The first parameter is format string describing how the rest of the parameters should be formatted. See the documentation for the `Kernel::sprintf` function in Ruby for all the details.

- Type: rvalue

tag

Add the specified tags to the containing class or definition. All contained objects will then acquire that tag, also.

- Type: statement

tagged

A boolean function that tells you whether the current container is tagged with the specified tags. The tags are ANDed, so that all of the specified tags must be included for the function to return true.

- Type: rvalue

template

Evaluate a template and return its value. See the templating docs for more information.

Note that if multiple templates are specified, their output is all concatenated and returned as the output of the function.

- Type: rvalue

versioncmp

Compares two version numbers.

Prototype:

```
$result = versioncmp(a, b)
```

Where a and b are arbitrary version strings.

This function returns:

- `1` if version a is greater than version b
- `0` if the versions are equal
- `-1` if version a is less than version b

Example:

```
if versioncmp('2.6-1', '2.4.5') > 0 {
    notice('2.6-1 is > than 2.4.5')
}
```

This function uses the same version comparison algorithm used by Puppet's `package` type.

- Type: rvalue

warning

Log a message on the server at level warning.

- Type: statement

This page autogenerated on Tue Jun 18 17:01:37 -0700 2013

Metaparameter Reference

Metaparameter Reference

This page is autogenerated; any changes will get overwritten (last generated on Tue Jun 18 17:01:52 -0700 2013)

Metaparameters

Metaparameters are parameters that work with any resource type; they are part of the Puppet framework itself rather than being part of the implementation of any given instance. Thus, any defined metaparameter can be used with any instance in your manifest, including defined□ components.

Available Metaparameters

alias

Creates an alias for the object. Puppet uses this internally when you provide a symbolic title:

```
file { 'sshdconfig':
  path => $operatingsystem ? {
    solaris => "/usr/local/etc/ssh/sshd_config",
    default => "/etc/ssh/sshd_config"
  },
  source => "..."
}

service { 'sshd':
  subscribe => File['sshdconfig']
}
```

When you use this feature, the parser sets `sshdconfig` as the title, and the library sets that as an alias for the file so the dependency lookup in `Service['sshd']` works. You can use this metaparameter yourself, but note that only the library can use these aliases; for instance, the following code will not work:

```
file { "/etc/ssh/sshd_config":
  owner => root,
  group => root,
  alias => 'sshdconfig'
}

file { 'sshdconfig':
  mode => 644
}
```

There's no way here for the Puppet parser to know that these two stanzas should be affecting the□ same file.□

See the Language Guide for more information.

audit

Marks a subset of this resource's unmanaged attributes for auditing. Accepts an attribute name, an array of attribute names, or `all`.

Auditing a resource attribute has two effects: First, whenever a catalog is applied with puppet apply or puppet agent, Puppet will check whether that attribute of the resource has been modified, comparing its current value to the previous run; any change will be logged alongside any actions performed by Puppet while applying the catalog.

Secondly, marking a resource attribute for auditing will include that attribute in inspection reports generated by puppet inspect; see the puppet inspect documentation for more details.

Managed attributes for a resource can also be audited, but note that changes made by Puppet will be logged as additional modifications. (I.e. if a user manually edits a file whose contents are audited and managed, puppet agent's next two runs will both log an audit notice: the first run will log the user's edit and then revert the file to the desired state, and the second run will log the edit made by Puppet.)

before

References to one or more objects that depend on this object. This parameter is the opposite of require — it guarantees that the specified object is applied later than the specifying object:

```
file { "/var/nagios/configuration":
  source  => "...",
  recurse => true,
  before  => Exec["nagios-rebuid"]
}

exec { "nagios-rebuild":
  command => "/usr/bin/make",
  cwd     => "/var/nagios/configuration"
}
```

This will make sure all of the files are up to date before the make command is run.

check

Audit specified attributes of resources over time, and report if any have changed. This parameter has been deprecated in favor of 'audit'.

loglevel

Sets the level that information will be logged. The log levels have the biggest impact when logs are sent to syslog (which is currently the default). Valid values are `debug`, `info`, `notice`, `warning`, `err`, `alert`, `emerg`, `crit`, `verbose`.

noop

Boolean flag indicating whether work should actually be done. Valid values are `true`, `false`.

notify

References to one or more objects that depend on this object. This parameter is the opposite of subscribe — it creates a dependency relationship like before, and also causes the dependent object(s) to be refreshed when this object is changed. For instance:

```
file { "/etc/sshd_config":
  source => "....",
  notify => Service['sshd']
}

service { 'sshd':
  ensure => running
}
```

This will restart the sshd service if the sshd config file changes.

require

References to one or more objects that this object depends on. This is used purely for guaranteeing that changes to required objects happen before the dependent object. For instance:

```
# Create the destination directory before you copy things down
file { "/usr/local/scripts":
  ensure => directory
}

file { "/usr/local/scripts/myscript":
  source  => "puppet://server/module/myscript",
  mode    => 755,
  require => File["/usr/local/scripts"]
}
```

Multiple dependencies can be specified by providing a comma-separated list of resources, enclosed in square brackets:

```
require => [ File["/usr/local"], File["/usr/local/scripts"] ]
```

Note that Puppet will autorequire everything that it can, and there are hooks in place so that it's easy for resources to add new ways to autorequire objects, so if you think Puppet could be smarter here, let us know.

In fact, the above code was redundant — Puppet will autorequire any parent directories that are being managed; it will automatically realize that the parent directory should be created before the script is pulled down.

Currently, exec resources will autorequire their CWD (if it is specified) plus any fully qualified paths

that appear in the command. For instance, if you had an `exec` command that ran the `myscript` mentioned above, the above code that pulls the file down would be automatically listed as a requirement to the `exec` code, so that you would always be running againts the most recent version.

schedule

On what schedule the object should be managed. You must create a schedule object, and then reference the name of that object to use that for your schedule:

```
schedule { 'daily':
  period => daily,
  range  => "2-4"
}

exec { "/usr/bin/apt-get update":
  schedule => 'daily'
}
```

The creation of the schedule object does not need to appear in the configuration before objects that use it.

stage

Which run stage a given resource should reside in. This just creates a dependency on or from the named milestone. For instance, saying that this is in the 'bootstrap' stage creates a dependency on the 'bootstrap' milestone.

By default, all classes get directly added to the 'main' stage. You can create new stages as resources:

```
stage { ['pre', 'post']: }
```

To order stages, use standard relationships:

```
stage { 'pre': before => Stage['main'] }
```

Or use the new relationship syntax:

```
Stage['pre'] -> Stage['main'] -> Stage['post']
```

Then use the new class parameters to specify a stage:

```
class { 'foo': stage => 'pre' }
```

Stages can only be set on classes, not individual resources. This will fail:

```
file { '/foo': stage => 'pre', ensure => file }
```

subscribe

References to one or more objects that this object depends on. This metaparameter creates a dependency relationship like require, and also causes the dependent object to be refreshed when the subscribed object is changed. For instance:

```
class nagios {
  file { 'nagconf':
    path   => "/etc/nagios/nagios.conf"
    source => "puppet://server/module/nagios.conf",
  }
  service { 'nagios':
    ensure    => running,
    subscribe => File['nagconf']
  }
}
```

Currently the `exec`, `mount` and `service` types support refreshing.

tag

Add the specified tags to the associated resource. While all resources are automatically tagged with as much information as possible (e.g., each class and definition containing the resource), it can be useful to add your own tags to a given resource.

Multiple tags can be specified as an array:

```
file {'/etc/hosts':
  ensure => file,
  source => 'puppet:///modules/site/hosts',
  mode   => 0644,
  tag    => ['bootstrap', 'minimumrun', 'mediumrun'],
}
```

Tags are useful for things like applying a subset of a host's configuration with the `tags` setting:

```
puppet agent --test --tags bootstrap
```

This way, you can easily isolate the portion of the configuration you're trying to test.

This page autogenerated on Tue Jun 18 17:01:53 -0700 2013

Report Reference

Report Reference

This page is autogenerated; any changes will get overwritten (last generated on Tue Jun 18 17:01:59 -0700 2013)

Puppet clients can report back to the server after each transaction. This transaction report is sent as a YAML dump of the `Puppet::Transaction::Report` class and includes every log message that was generated during the transaction along with as many metrics as Puppet knows how to collect. See Reports and Reporting for more information on how to use reports.

Currently, clients default to not sending in reports; you can enable reporting by setting the `report` parameter to true.

To use a report, set the `reports` parameter on the server; multiple reports must be comma-separated. You can also specify `none` to disable reports entirely.

Puppet provides multiple report handlers that will process client reports:

http

Send report information via HTTP to the `reporturl`. Each host sends its report as a YAML dump and this sends this YAML to a client via HTTP POST. The YAML is the body of the request.

log

Send all received logs to the local log destinations. Usually the log destination is syslog.

rrdgraph

Graph all available data about hosts using the RRD library. You must have the Ruby RRDtool library installed to use this report, which you can get from the RubyRRDTool RubyForge page.
This package may also be available as `ruby-rrd` or `rrdtool-ruby` in your distribution's package management system. The library and/or package will both require the binary `rrdtool` package from your distribution to be installed.

This report will create, manage, and graph RRD database files for each of the metrics generated during transactions, and it will create a few simple html files to display the reporting host's graphs. At this point, it will not create a common index file to display links to all hosts.

All RRD files and graphs get created in the `rrddir` directory. If you want to serve these publicly, you should be able to just alias that directory in a web server.

If you really know what you're doing, you can tune the `rrdinterval`, which defaults to the `runinterval`.

store

Store the yaml report on disk. Each host sends its report as a YAML dump and this just stores the file on disk, in the `reportdir` directory.

These files collect quickly – one every half hour – so it is a good idea to perform some maintenance on them if you use this report (it's the only default report).

tagmail

This report sends specific log messages to specific email addresses based on the tags in the log messages.

See the documentation on tags for more information.

To use this report, you must create a `tagmail.conf` file in the location specified by the `tagmap` setting. This is a simple file that maps tags to email addresses: Any log messages in the report that match the specified tags will be sent to the specified email addresses.

Lines in the `tagmail.conf` file consist of a comma-separated list of tags, a colon, and a comma-separated list of email addresses. Tags can be !negated with a leading exclamation mark, which will subtract any messages with that tag from the set of events handled by that line.

Puppet's log levels (`debug`, `info`, `notice`, `warning`, `err`, `alert`, `emerg`, `crit`, and `verbose`) can also be used as tags, and there is an `all` tag that will always match all log messages.

An example `tagmail.conf`:

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

This will send all messages to `me@domain.com`, and all messages from webservers that are not also from mailservers to `httpadmins@domain.com`.

If you are using anti-spam controls such as grey-listing on your mail server, you should whitelist the sending email address (controlled by `reportform` configuration option) to ensure your email is not discarded as spam.

This page autogenerated on Tue Jun 18 17:01:59 -0700 2013