# Open source Puppet documentation

# Contents

# Welcome to Puppet 6

Puppet provides tools to automate managing your infrastructure. Puppet is an open source product with a vibrant community of users and contributors. You can get involved by fixing bugs, influencing new feature direction, publishing your modules, and engaging with the community to share knowledge and expertise.

| Helpful Puppet docs links | | Other useful places | |
|---|---|---|---|
| **What is in Puppet?** | Overview of Puppet's architecture<br>Hiera<br>Facter<br>Puppet Server<br>PuppetDB | **Other Puppet open source docs and projects** | Puppet Development Kit docs<br>Bolt docs<br>Open source projects on Github |
| **Installing and upgrading** | Release notes<br>System requirements<br>Install Puppet<br>Install agents<br>Configuring<br>Upgrading | **Get involved** | Forge<br>Community |
| | | **Learn more about Puppet** | Blog posts about open source Puppet<br>Puppet training<br>Learning roadmap |
| **The details** | Resource type reference and cheat sheet<br>Puppet language<br>Language visual index<br>Language style guide | **Need more?** | Try Puppet Enterprise<br>Puppet Enterprise docs<br>Continuous Delivery for Puppet Enterprise docs |
| **Modules** | Fundamentals<br>Beginner's guide<br>Puppet Strings<br>Puppet Strings style guide | **Select Puppet docs in other languages** | Japanese (日本語)<br>Spanish (Español)<br>German (Deutsch) |

To send us feedback or let us know about a docs error: rate the page out of five stars and provide details in the feedback form. Or you can open a ticket (you'll need a Jira account).

## Puppet architecture

Puppet is configured in an agent-master architecture, in which a *master* node controls configuration information for a fleet of managed *agent* nodes.

**Note:** Previous versions of Puppet offered a *standalone architecture*, in which agents compiled their own catalogs using the Puppet apply application. We no longer recommend this configuration, because it's challenging to maintain and secure.

Master-agent communication follows this pattern:

1. An agent node sends facts to the master and requests a catalog.
2. The master compiles and returns the node’s catalog using the sources of information the master has access to.

3. The agent applies the catalog to the node by checking each resource the catalog describes. If it finds resources that are not in their desired state, it makes the changes necessary to correct them. Or, in no-op mode, it assesses what changes would be needed to reconcile the catalog.
4. The agent sends a report back to the master.

### Communication and security in agent-master installations

Masters and agents communicate by HTTPS using SSL certificates.

Puppet includes a built-in certificate authority for managing certificates. Agents automatically request certificates through the master's HTTP endpoint, and you use the `puppetserver ca` command to inspect requests and sign new certificates.

## The `puppet-agent` package and version numbers

The `puppet-agent` package contains multiple components. Understanding how package versions are numbered and which versions go with each other is important when upgrading and troubleshooting.

### Component version numbers

This table shows which components shipped in which `puppet-agent` release and contains links to available component release notes. In version 6.1 and later, agent release notes are included on the same page as Puppet release notes.

**Note:** Hiera 5 is a backward-compatible evolution of Hiera, which is built into Puppet. To provide some backward-compatible features, it uses the classic Hiera 3 codebase. This means that Hiera is still shown as version 3.x in the table above, even though this Puppet version uses Hiera 5.

| `puppet-agent` | Puppet | Facter | Hiera | Resource API | Ruby | OpenSSL |
|---|---|---|---|---|---|---|
| 6.7.2 | 6.7.2 | 3.14.2 | 3.5.0 | 1.8.6 | 2.5.3 | 1.1.1a |
| 6.7.1 | This version of Puppet was never released. | | | | | |
| 6.7.0 | 6.7.0 | 3.14.2 | 3.5.0 | 1.8.6 | 2.5.3 | 1.1.1a |
| 6.6.0 | 6.6.0 | 3.14.1 | 3.5.0 | 1.8.5 | 2.5.3 | 1.1.1a |
| 6.5.0 | 6.5.0 | 3.14.0 | 3.5.0 | 1.8.4 | 2.5.3 | 1.1.1a |
| 6.4.2 | 6.4.2 | 3.13.2 | 3.5.0 | 1.8.3 | 2.5.3 | 1.1.1a |
| 6.4.1 | 6.4.1 | 3.13.2 | 3.5.0 | 1.8.2 | 2.5.3 | 1.1.1a |
| 6.4.0 | 6.4.0 | 3.13.1 | 3.5.0 | 1.8.1 | 2.5.3 | 1.1.1a |
| 6.3.0 | 6.3.0 | 3.13.0 | 3.5.0 | 1.7.0 | 2.5.3 | 1.1.1a |
| 6.2.0 | 6.2.0 | 3.12.3 | 3.5.0 | 1.6.3 | 2.5.3 | 1.1.1a |
| 6.1.0 | 6.1.0 | 3.12.2 | 3.5.0 | 1.6.2 | 2.5.3 | 1.1.0h |
| 6.0.8 | 6.0.8 | 3.12.4 | 3.4.6 | 1.6.4 | 2.5.3 | 1.1.1a |
| 6.0.7 | 6.0.7 | 3.12.4 | 3.4.6 | 1.6.4 | 2.5.3 | 1.1.1a |
| 6.0.6 | This version of Puppet was never released. | | | | | |
| 6.0.5 | 6.0.5 | 3.12.3 | 3.4.6 | 1.6.3 | 2.5.3 | 1.1.1a |
| 6.0.4 | 6.0.4 | 3.12.1 | 3.4.5 | 1.6.2 | 2.5.2 | 1.1.0h |
| 6.0.3 | 6.0.3 | 3.12.1 | 3.4.5 | 1.6.0 | 2.5.1 | 1.1.0h |

| `puppet-agent` | Puppet | Facter | Hiera | Resource API | Ruby | OpenSSL |
|---|---|---|---|---|---|---|
| 6.0.2 | 6.0.2 | 3.12.0 | 3.4.5 | 1.6.0 | 2.5.1 | 1.1.0h |
| 6.0.1 | 6.0.1 | 3.12.0 | 3.4.5 | 1.5.0 | 2.5.1 | 1.1.0h |
| 6.0.0 | 6.0.0 | 3.12.0 | 3.4.5 | 1.5.0 | 2.5.1 | 1.1.0h |

### What `puppet-agent` and `puppetserver` are

We distribute Puppet as two core packages.

**`puppet-agent`**
This package contains Puppet's main code and all of the dependencies needed to run it, including Facter, Hiera, and bundled versions of Ruby and OpenSSL. After it's installed, you have everything you need to run the Puppet agent service and the `puppet apply`command.

**`puppetserver`**
This package depends on `puppet-agent` and adds the JVM-based Puppet Server application. After it's installed, Puppet Server can serve catalogs to nodes running the agent service.

**Note:** As of Puppet agent 5.5.4, MCollective was deprecated. It was removed in Puppet agent 6.0. If you use Puppet Enterprise, consider Puppet . If you use open source Puppet, migrate MCollective agents and filters using tools such as Bolt and PuppetDB's Puppet Query Language.

### How Puppet version numbers work

The `puppet-agent` package distributes several different pieces of software, some of which have different version numbers. For example `puppet-agent` 6.3.0 shipped Puppet version 6.3.0, but Facter version 3.13.0 and Puppet Server 6.2.0. The `puppet-agent` package doesn't necessarily match the version of Puppet it installs.

Puppet Server is a separate application that, among other things, runs instances of the Puppet master application. It has its own version number separate from the version of Puppet it runs and might be compatible with more than one existing Puppet version.

Upgrade order is important. First, update Puppet Server, then `puppet-agent`. If you upgrade Puppet Server or PuppetDB to version 5 or newer, it automatically upgrades the `puppet-agent` package on the master to version 5.0.0 or newer. Puppet Server 5 also restricts you from installing anything older than `puppet-agent` 5.0.0 on your agent nodes.

## Master and agent compatibility

Use this table to verify that you're using a compatible version of the agent for your PE or Puppet master.

| | | PE 3.x<br>Puppet 3.x |
|---|---|---|
| Agent | 3.x | ✓ |
| | 4.x | |
| | 5.x | |
| | 6.x | |

**Note:**

- Puppet 3.x has reached end of life and is not actively developed or tested. We retain agent 3.x compabilility with later versions of the master only to enable upgrades.
- You can use pre-6.x agents with a Puppet 6.x or PE 2019.0 or later master, but this combination doesn't take advantage of the new intermediate certificate authority architecture introduced in Puppet Server 6.0. To adopt the new CA architecture, both your master and agents must be upgraded to 6.x/2019.0, and you must regenerate certificates. If you don't upgrade *all* of your nodes to 6.x, don't regenerate certificates, because pre-6.x agents won't work with the new CA architecture.

# Puppet language and code

You'll use Puppet's declarative language to describe the desired state of your system in files called manifests. Manifests describe how your network and operating system resources, such as files, packages, and services, should be configured. Puppet then compiles those manifests into catalogs, which across your infrastructure.

### Related information

The Puppet language on page 171
You'll use Puppet's declarative language to describe the desired state of your system in files called manifests. Manifests describe how your network and operating system resources, such as files, packages, and services, should be configured. Puppet then compiles those manifests into catalogs, and applies each catalog to its corresponding node to ensure the node is configured correctly, across your infrastructure.

Language visual index on page 173
A quick reference of syntax examples for the Puppet language.

The Puppet language style guide on page 178
This style guide promotes consistent formatting in the Puppet language, giving you a common pattern, design, and style to follow when developing modules. This consistency in code and module structure makes it easier to update and maintain the code.

## Catalogs

To configure nodes, the Puppet master compiles configuration information into a catalog, which describes the desired state of a specific agent node. Each agent requests and receives its own individual catalog.

The catalog describes the desired state for every resource that should be managed on a single given node. Whereas a manifest can contain conditional logic to describe specific resource configuration for multiple nodes, a catalog is a static document that describes all of the resources and and dependencies for only one node.

To create a catalog for a given agent, the master compiles:

- Data from the agent, such as facts or certificates.
- External data, such as values from functions or classification information from the PE console.
- Manifests, which can contain conditional logic to describe the desired state of resources for many nodes.

The master resolves all of these elements and compiles a specific catalog for each individual agent. After the agent receives its catalog, it applies any changes needed to bring the agent to the state described in the catalog.

**Tip:** When you run the `puppet apply` command on a node, it compiles the catalog locally and applies it immediately on the node where you ran the command.

Agents cache their most recent catalog. If they request a catalog and the master fails to compile one, they fall back to their cached catalog. For detailed information on the catalog compilation process, see the catalog compilation system page.

### Related information

Catalog compilation on page 504

When configuring a node, the agent uses a document called a catalog, which it downloads from the master. For each resource under management, the catalog describes its desired state and can specify ordered dependency information.

## Resources and classes

A resource describes some aspect of a system, such as a specific service or package. You can group resources together in classes, which generally configure larger chunks of functionality, such as all of the packages, configuration files, and services needed to run an application.

The Puppet language is structured around resource declaration. When you declare a resource, you tell Puppet the desired state for that resource, so that Puppet can add it to the catalog and manage it. Every other part of the Puppet language exists to add flexibility and convenience to the way you declare resources.

Just as you declare a single resource, you can declare a class to manage many resources at once. Whereas a resource declaration might manage the state of a single file or package, a class declaration can manage everything needed to configure an entire service or application, including packages, configuration files, service daemons, and maintenance tasks. In turn, small classes that manage a few resources can be combined into larger classes that describe entire custom system roles, such as "database server" or "web application worker."

To add a class's resources to the catalog, either declare the class in a manifest or classify your nodes. Node classification allows you to assign a different set of classes to each node, based on the node's role in your infrastructure. You can classify nodes with node definitions or by using node-specific data from outside your manifests, such as that from an external node classifier or Hiera.

### Related information

Resources on page 207
Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.

Classes on page 220
*Classes* are named blocks of Puppet code that are stored in modules and applied later when they are invoked by name. You can add classes to a node’s catalog by either declaring them in your manifests or assigning them from an external node classifier (ENC). Classes generally configure large or medium-sized chunks of functionality, such as all of the packages, configuration files, and services needed to run an application.

## Manifests

Resources are declared in manifests, Puppet language files that describe how the resources should be configured. Manifests are a basic building block of Puppet and are kept in a specific file structure called a module. You can write your own manifests and modules or download them from Puppet or other Puppet users.

Manifests can contain conditional logic and declare resources for multiple agents. The master evaluates the contents of all the relevant manifests, resolves any logic, and compiles catalogs. Each catalog defines state for one specific node.

Manifests:

- Are text files with a `.pp` extension.
- Must use UTF-8 encoding.
- Can use Unix (LF) or Windows (CRLF) line breaks.

When compiling the catalog, the master always evaluates the main manifest first. This manifest, also known as the site manifest, defines global system configurations, such as LDAP configuration, DNS servers, or other configurations that apply to every node. The main manifest can be either a single manifest, usually named `site.pp`, or a directory containing several manifests, which Puppet treats as a single file. For more details about the main manifest, see the main manifest page.

The simplest Puppet deployment consists of a single main manifest file with a few resources. As you're ready, you can add complexity progressively, by grouping resources into modules and classifying your nodes more granularly.

### Manifest example

This short manifest manages NTP. It includes:

- A case statement that sets the name of the NTP service, depending on which operating system is installed on the agent.
- A `package` resource that installs the NTP package on the agent.
- A `service` resource that enables and runs the NTP service. This resource also applies the NTP configuration settings from `ntp.conf` to the service.
- A `file` resource that creates the `ntp.conf` file on the agent in `/etc/ntp.conf`. This resource also requires that the `ntp` package is installed on the agent. The contents of the `ntp.conf` file will be taken from the specified source file, which is contained in the `ntp` 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/code/modules/ntp/files/ntp.conf
}
```

# Glossary

## abstract data type

See type (data).

## agent

1. The agent software package, also known as `puppet-agent`. Puppet is usually deployed in a client-server arrangement. When you install Puppet on a device, you’re installing the agent software and its associated tools and dependencies. On *nix systems, this package is called `puppet-agent`.
2. The `puppet agent` daemon. You can invoke the agent software by running the `puppet agent` command, or you can let it run in the background as a daemon.
3. A *node* that is running the agent software. By association, a *node* running the agent daemon can be referred to as an agent node, or an agent.

An agent regularly performs *Puppet runs*, wherein it sends *facts* to a *master* and receives a configuration *catalog*, then applies the catalog to the local system using its *providers*.

**Note:** Puppet masters also run the agent daemon, which allows Puppet to manage its own configuration. In this sense, a master can also be an agent. An agent can also serve as its own master; this is often called a *masterless* configuration.

For more information about Puppet's agent-master architecture, see Puppet architecture.

## attribute

*Attributes* specify the desired state of a given configuration resource. Each resource type has a different set of attributes, and each attribute has its own set of allowed *values*. For example, a package resource (like `vim`) would have an `ensure` attribute, whose value could be `present`, `latest`, `absent`, or a version number:

```
package {'vim':
  ensure   => present,
  provider => apt,
}
```

You specify an attribute's value with the `=>` operator, and pairs of attributes and values are separated by commas.

## catalog

A *catalog* is a file that describes the desired state of each managed resource on a node. It is a *compilation* of all the *resources* that the Puppet *agent* applies to a given *node*, as well as the relationships between those resources.

Catalogs are compiled by a Puppet *master* from *manifests* and agent-provided data (such as *facts*, certificates, and an *environment* if one is provided), as well as optional external data (such as data from an *external node classifier*, *exported resources*, and *functions*). The master then serves the compiled catalog to the agent when requested.

Unlike the manifests from which they were compiled, catalogs don't contain any conditional logic or *functions*. They are unambiguous, relevant to only a specific node, and generated by a *node*.

For detailed information, see the Catalog compilation page in the Puppet documentation. See also *node graph*, a tool in *Puppet Enterprise* that interactively visualizes a node's catalog.

## class

A *class* is a collection of related *resources* that, after it's *defined*, can be *declared* as a single unit. For example, a class can contain all of the resources (such as files, settings, *modules*, and scripts) needed to configure the Apache webserver on a host. Classes can also declare other classes. For more information, see the Classes page in the *Puppet language* reference.

Classes are *singletons* and can be applied only one time in a given configuration; although the `include` function allows you to declare a class multiple times, Puppet evaluates it only one time.

The `require` and `contain` functions can create *relationships* between classes. For more information, see the Containment of resources page in the Puppet language reference.

**Note:** Being singletons, Puppet classes are not analogous to classes in object-oriented programming languages. Object-oriented classes are like templates that can be instantiated multiple times; Puppet's equivalent to this concept is *defined types*.

## classify

(or *node classification*)

To assign *classes* to a *node*, as well as provide any data required by the classes, you *classify* the node.

By writing a class, you enable a set of configurations. By classifying a node with the class, you describe the node's desired configuration.

You can classify nodes through *node definitions* in the *main manifest*, or with the Puppet Enterprise node manager or an *external node classifier*.

## compile

A *master* creates a *catalog* by *compilingmanifests* and agent-provided data, as well as any provided external data. During a *Puppet run*, an agent requests its *node*'s compiled catalog, then applies it to configure the node. For detailed information, see Catalog compilation.

## console

*Puppet Enterprise* includes a web user interface called the *console*, which provides tools for managing your Puppet infrastructure. For example, you can use it to *classify nodes*, manage Puppet services, trigger and view *reports* and metrics about *Puppet runs* and activity, and examine inventory data and *resources*. For detailed information, see the Puppet Enterprise documentation.

## core data type

(or *concrete data type*)

See type (data).

## data type

See type (data).

## declare

To direct Puppet to include a given *class* or *resource* in a given configuration, you *declare* it using the *Puppet language*. To declare classes, use a *function* (such as `include`) or the `class {'my_class':}` syntax. To declare resources, use the lowercase `file {'/tmp/my_file':}` syntax.

You can configure a resource or class when you declare it by including *attribute-value pairs*.

For more information, see Declaring classes and Resources. Contrast with *define*.

## define

1. *verb:* To specify the contents and behavior of a *class* or a *defined type*, you *define* it using the *Puppet language*. Defining a class or type doesn't automatically include it in a configuration; it simply makes it available to be *declared*.
2. *noun:* an older term for a *defined type*. For instance, a module might refer to its available defines.
3. *keyword:* You use the `define` Puppet language keyword to create a defined type.

For more information, see Defined types.

## defined type

(or *defined resource type*)

See type (defined).

## design pattern

A *design pattern* is a colloquial term used to describe a collection of related *manifests* that are designed to solve an issue or manage a particular configuration item. For example, an "Apache pattern" refers to the manifests designed to configure Apache. See also module.

## environment

An *environment* is an isolated group of *agent nodes* that a *master* can serve with its own *main manifest* and set of *modules*. For example, you can use environments to set up scratch *nodes* for testing before rolling out changes to production, or divide a site by types of hardware.

For more information, see About environments.

## exported resource

An *exported resource* is a *resource* that you've *declared* to be available to other *nodes* that can then collect the exported resource and manage their own copies. This lets you share a resource's desired state across nodes, such as when one node depends on information on another node for its configuration, or when you need to monitor a resource's state.

For more information, see Exported resources.

## expression

The Puppet language supports several types of *expressions* for comparison and evaluation purposes, including Boolean expressions, comparison expressions, and arithmetic expressions. For more information, see Expressions and operators.

## external node classifier

An *external node classifier* (ENC) is an executable script that returns information about which *classes* to apply to a *node* when called by a *master*. For example, Puppet Enterprise acts as an ENC.

ENCs provide an alternative to using a site's *main manifest* to classify nodes. An ENC can be written in any language, and can use information from any data source (such as an LDAP database) when classifying nodes.

An ENC is called with the name of the node to be classified as an argument, and returns a YAML document describing the node. For more information, see Writing external node classifiers.

## fact

A *fact* is a piece of information about a *node*, such as its hostname, IP address, and operating system.

*Facter* reads facts from a node and makes them available to Puppet. You can extend Facter with custom facts, which can expose *site-specific* details of your systems to your Puppet *manifests*. For more information, see Custom facts.

## Facter

*Facter* is Puppet's system inventory tool. Facter reads *facts* about a *node*, such as its hostname, IP address, and operating system, and makes them available to Puppet.

Facter includes many built-in facts, and you can view their names and *values* for the local system by running `facter` at a node's command line.

In agent-master Puppet arrangements, Puppet agents send their nodes' facts to the Puppet master.

For more information, see the Facter documentation.

## filebucket

A repository in which Puppet stores file backups when it has to replace files is called a *filebucket*.

A filebucket can be local (and owned by the *node* being managed) or site-global (and owned by the *master*). Typically, a single filebucket is defined for a network and is used as the default backup location. For more information, see filebucket type.

## function

A *function* is a *Puppet language* statement that returns a *value* or modifies a *catalog*. Puppet has many built-in functions, and Puppet *modules* can add their own functions. You can also write custom functions.

Functions generally take at least one value as an argument, execute Puppet code, and return a value. Because Puppet evaluates functions during *compilation*, the Puppet master executes them in an agent-master arrangement. Puppet functions can only access the *facts* that an agent submitted.

Contrast with lambda.

## global scope

See scope.

## Hiera

The *Hiera* tool, which ships with the *agent*, provides hierarchical key-value lookup for *site-specific* data. This lets you take site-specific data out of your *manifests* and place it in a centralized location, while also setting default data and overriding certain specified *values* when necessary. For more information, see the Hiera documentation.

## host

1. Any device, physical or virtual, attached to a network is a *host*. In the Puppet documentation, this can also refer to a device running the *agent* daemon. See also node.
2. *resource type:* A *host* can refer to an entry in a system's `hosts` file, used for name resolution. Puppet includes a *type module* to manage hosts. For more information, see the Host core module on the Forge.

## idempotent

*Idempotence* refers to the concept of doing something multiple times with the same outcome. Puppet *resources* are idempotent, because they describe a desired final state rather than a series of steps to follow.

The most prominent exception among Puppet resources is the `exec` resource type, which is idempotent but relies on the user to design them accordingly.

## inheritance (class)

A Puppet *class* can be derived from another class by using the `inherits` keyword. The derived class *declares* all of the same *resources*, but can override some of their *attributes* and add new resources. Use inheritance very sparingly. for more information, see Classes.

## lambda

(or *code block*)

A *lambda* is a block of parameterized *Puppet language* code that you can pass to certain *functions*. For more information, see Lambdas.

## local scope

See scope.

## main manifest

(or *site manifest*)

The *main manifest* is the main point-of-entry *manifest* used by a *master* when *compiling* a *catalog*.

The site manifest file is usually named `site.pp`. Its location is set with the `manifest` setting in `environment.conf`, with a default location set by the `default_manifest` setting in `puppet.conf`.

For more information, see Main manifest directory.

## manifest

(or *Puppet code*)

A *manifest* file contains code written in the *Puppet language* and is named with the `.pp` file extension. The Puppet code in a manifest can:

- *Declare resources* and *classes*
- Set *variables*
- Evaluate *functions*
- *Define classes*, *defined types*, *functions*, and *nodes*

Most manifests are contained in *modules*. Every manifest in a module defines a single class, defined type, or function.

The *master* service reads an *environment*'s *main manifest*. This manifest usually defines *nodes*, so that each managed *agent* receives a unique *catalog*.

See also main manifest.

## manifest ordering

Puppet uses *manifest ordering* to apply unrelated *resources* in the order that they're declared in their *manifests*. For related resources, or for more information about manifest ordering, see Relationships and ordering.

## master

(or *Puppet master*)

In a standard Puppet client-server deployment, the server is known as the *master*. The master *compiles* and serves configuration *catalogs* on demand to *agents* on client *nodes*.

The master provides catalogs to agents using an HTTP server. It can run as a standalone daemon with a built-in web server, or as part of *Puppet Server*. (We don't recommend using the built-in daemon when managing more than 10 nodes.)

For more information, see Overview of Puppet's architecture.

## masterless

*Masterless* refers to a Puppet *node* that does not rely on a *master* for its *catalog*. See agent.

## metaparameter

A *metaparameter* is a *resource attribute* that can be specified for any type of resource. Metaparameters are part of Puppet's framework rather than part of a specific *type*, and usually affect the way resources relate to each other. For a list of metaparameters and their usage, see Metaparameter reference.

## module

A *module* is a collection of *classes*, *resource types*, files, *functions* and *templates*, organized around a particular purpose. For example, a module can configure an Apache web server instance or Rails application. There are many modules available for download in the Puppet Forge.

For more information, see Module fundamentals and Installing modules.

## namevar

(or *name*)

The *namevar* attribute represents a *resource*'s unique identity on the target system. For example, two different files cannot have the same `path`, and two different services cannot have the same `name`.

Every resource *type* has a designated namevar, usually `name`. Some types, such as `file` or `exec`, have their own (in these cases, `path` and `command`, respectively). If a type's namevar is something other than `name`, it's mentioned in the type reference documentation.

If you don't specify a *value* for a resource's namevar when you *declare* it, it defaults to that resource's *title*.

## no-op

(or *noop*)

By running Puppet in *no-op* mode (`noop` in code, short for "no operations" mode), you can simulate what Puppet will do without actually changing anything. No-op mode allows you to perform a *dry run* that logs planned activity but doesn't affect any *nodes*. To run Puppet in no-op mode, run `puppet agent` or `puppet apply` with the `--noop` flag.

## node

A *node* is a device managed by Puppet. Some nodes are *masters*, which compile *manifests* into *catalogs*; most nodes, including most masters, are *agents*, which receive catalogs and apply them to the node during a *Puppet run*. Most nodes are computers (such as workstations and servers), but some aren't (such as supported network switches and storage appliances).

A node is also one of the fundamental units of a *puppetized* infrastructure. See also classify, node definition, and scope.

## node definition

(or *node statement*)

A *node definition* is a collection of *classes*, *resources*, and *variables* in a *manifest* that are only applied to a certain *agent node*. Node definitions begin with the `node` keyword, and can match a node by full name or regular expression.

When a node retrieves or *compiles* its *catalog*, it receives the contents of a single matching node statement as well as any classes or resources declared outside any node statement. The classes in every other node statement are hidden from that node.

For more information, see Node definitions.

## node graph

The *node graph* in Puppet Enterprise provides a graphic representation of a *node*'s *resources*. For more information, see Inspecting your infrastructure in the Puppet Enterprise documentation.

## node management

Puppet Enterprise provides *node management* workflows and tools to help you classify nodes using fact-based rules in the console. For more information, see Managing nodes in the Puppet Enterprise documentation.

## node run status

When Puppet manages a *node* during a *Puppet run*, it attempts bring it into compliance with its *catalog*. The state of the node after a run, as tracked by Puppet Enterprise, is called the *node run status*. The following node run statuses are possible:

- With failures
- With corrective changes
- With intentional changes
- Unchanged

When a no-op Puppet run determines that it would have modified the node during a normal Puppet run, the node run status indicates that the change would have been made (but wasn't). There are also run statuses for unresponsive or not completely configured nodes.

For more information, see Monitoring current infrastructure state in the Puppet Enterprise documentation.

## node scope

See scope.

## notification

A *notification* is a type of *relationship* that both declares an *order* for resources and causes *refresh* events to be sent. A notification relationship is set with the `notify` metaparameter or the wavy chaining arrow `~>`. For more information, see Relationships and ordering.

## ordering

By *ordering* resources, you determine which *resources* are managed before others.

By default, Puppet uses *manifest ordering*, which evaluates resources in the same order they're declared in their *manifests*. Puppet also obeys *relationships* you provide that determine whether a resource depends on other resources. For more information, see Relationships and ordering.

## parameter

A *parameter* is a chunk of information that a *class* or *resource* can accept.

1. *In custom type and provider development:* a parameter does not call a method on a *provider*. They are eventually expressed as *attributes* in instances of this *resource type*. For more information, see Custom types development.
2. *In defined types and parameterized classes:* a parameter is a *variable* in the *definition* of a *class* or *defined type*, whose *value* is set by a *resource attribute* when an instance of that type or class is *declared*.

```
define my_new_type ($my_parameter) {
  file { "$title":
    ensure  => file,
    content => $my_parameter,
  }
}

my_new_type { '/tmp/test_file':
  my_parameter => "This text will become the content of the file.",
}
```

   The parameters you use when defining a type or class define the attributes available when the type or class is declared.
3. *For external nodes:* A parameter is a top-scope *variable* set by an *external node classifier*. Although these are called parameters, they are just normal variables; the name refers to how they are usually used to configure the behavior of *classes*.

## plug-in

A *plug-in* is a custom *type*, *function*, or *fact* that extends Puppet's capabilities and is distributed in a *module*. For more information, see Plug-ins in modules.

## plusignment operator

The *plusignment operator* `+>` adds *values* to *resource attributes* using the plusignment syntax. This is useful when you want to override resource attributes without having to specify already declared values a second time. For more information, see the section about appending to resource attributes in Classes.

## profile

In the *roles and profiles* pattern of Puppet code development, a *profile* represents the configuration of a technology stack for a *site*, and typically consists of one or more *classes*. A role can include as many profiles as required to define itself. Profiles are included in *role and profile modules*.

For more information, see The roles and profiles method in the Puppet Enterprise documentation.

## property

In custom type and provider development, a *property* is a *value* that corresponds to an observable part of the target *node*'s state. When retrieving a *resource*'s state, a property calls the specified method on the *provider*, which reads the state from the system. If the current state does not match the specified state, the provider changes it.

Properties appear as *attributes* when *declaring* instances of this resource *type*. For more information, see Custom types development.

## provider

*Providers* implement *resource types* on a specific type of system by using the system's own tools. The division between types and providers allows a single resource type (such as `package`) to manage packages on many different systems by using, for example, `yum` on Red Hat systems, `dpkg` and `apt` on Debian-based systems, and `ports` on BSD systems.

Providers are often Ruby wrappers around shell commands, and can be relatively straightforward to create.

## Puppet

*Puppet* can refer to several things:

- The Puppet suite of automation products.
- The open source Puppet project.
- The command you run to invoke the Puppet *agent* daemon on a *node*.
- The *Puppet language* that you use you write *manifests*.
- The company, Puppet, Inc.

## Puppet Enterprise

*Puppet Enterprise* is a distribution for the Puppet family of systems automation tools. It includes several tools for managing Puppet across thousands of *nodes*, such as the console, node and configuration management views, code management, Razor, and access control. For more information, see the Puppet Enterprise documentation.

## Puppet language

You write Puppet code in the *Puppet language*. Puppet language files are called *manifests* and are named with the `.pp` extension. The *master compiles* this Puppet code into a *catalog* during a *Puppet run*. For more information, see Basics and the rest of the Puppet language docs.

## Puppet run

A *Puppet run* is when an *agent* sends *facts* and an identifying certificate to the *master*, and requests a *compiled catalog* in return. The agent applies that catalog to the *node* by using operating system-specific *providers* to bring the node's *properties* in line with the catalog's definitions, then sends a *report* of logs and metrics to the master.

By default, a Puppet run takes place every 30 minutes, even when an agent's catalog or configuration haven't changed. For a detailed description of a Puppet run, see Agent-master HTTPS communications.

## Puppet Server

*Puppet Server* is an open-source Java Virtual Machine application that provides high-performance, scalable *master* services and an administrative API for maintenance. For more information, see the Puppet Server documentation.

## PuppetDB

*PuppetDB* is an open-source database that caches and stores data generated by Puppet. This makes Puppet work faster and provides an API for other applications to access Puppet's collected data. It also enables advanced Puppet features, such as *exported resources*. For more information, see the PuppetDB documentation.

## Puppetfile

A *Puppetfile* is an authoritative, standalone list that tells *r10k* which *modules* to install, what versions to install, and which source to use. This lets you manage the installation of sets of modules. For more information, see Managing environment content with a Puppetfile in the Puppet Enterprise docs.

## puppetize

To *puppetize* your infrastructure is to bring it under automated management by capturing it in *Puppet language* form and using Puppet to reach and maintain the desired state. Puppetizing your infrastructure is an iterative, incremental, and ongoing process of gradually bringing more and more of your services, applications, nodes, and devices under Puppet management.

## r10k

The *r10k* tool in *Puppet Enterprise* helps you manage Puppet code in *environments* and *modules* by using *Puppetfiles*. For more information, see Managing and deploying Puppet code in the Puppet Enterprise docs.

## Razor

*Razor* is a *Puppet Enterprise* provisioning application designed to help you discover, configure, and deploy bare-metal hardware, even if it doesn't yet have an operating system. For more information, see Provisioning with Razor in the Puppet Enterprise docs.

## realize

To specify that a *virtual resource* is applied to the current system, it must be *realized*. After you *declare* a virtual resource, there are two methods for realizing it:

- Use the spaceship syntax `<| |>`
- Use the `realize` *function*.

A virtually declared resource is present in the *catalog* but won't be applied to a system until it is realized. For more information, see Virtual resources.

## refresh

A *resource* is *refreshed* when a resource it *subscribes* to (or which *notifies* it) is modified.

Different resource types do different things when they're refreshed. For instance, services restart, mount points unmount and remount, and `exec`s execute if the `refreshonly` attribute is set.

## relationship

A rule that sets the order in which *resources* are managed creates a *relationship* between those resources. For more information, see Relationships and ordering.

## report

You can configure Puppet *agents* to send *reports* containing logs and metrics at the end of every *Puppet run*. A *report processor* transforms those reports into a different format and sends it to another application, location, or service.

For more information, see Tracking Puppet activity with reports.

## report processor

A *report processor* takes a Puppet *report*, transforms it to a specific format, and sends it to another application, location, or service. Puppet ships with built-in report processors, and you can write your own. For more information, see Tracking Puppet activity with reports.

## resource

A *resource* is a unit of configuration whose state can be managed by Puppet. Every resource has a *type* (such as `file`, `service`, or `user`), a *title*, and one or more *attributes* with specified *values*.

Resources can be large or small, and simple or complex. They do not always directly map to simple details on the client — they might involve spreading information across multiple files or modifying devices. For example, a `service` resource only models a single service, but might involve executing an init script, running an external command to check its status, and modifying the system's run level configuration.

For more information about resources, see Resources.

## resource declaration

A *resource declaration* is a fragment of Puppet code that details the desired state of a *resource* and instructs Puppet to manage it. This term helps to differentiate between the literal resource on a system and the specification for how to manage that resource. However, resource declarations are often referred to simply as *resources*.

## role

In the *roles and profiles* pattern of Puppet coding, a *role* defines the business purpose that a *node* performs. A role typically consists of one *class* that can completely configure categories of nodes with *profiles*. We recommend that a node only has one role; if a node requires more than one existing role, create a new role for it.

For more information, see Roles and profiles in the Puppet Enterprise documentation.

## role and profile module

A *role and profile module* is a Puppet *module* that assigns configuration data to groups of *nodes* based on *roles* and *profiles*. A role and profile module doesn't have any special features, but it represents an abstract, private, *site-specific* way to use modules to configure technology stacks and node descriptions.

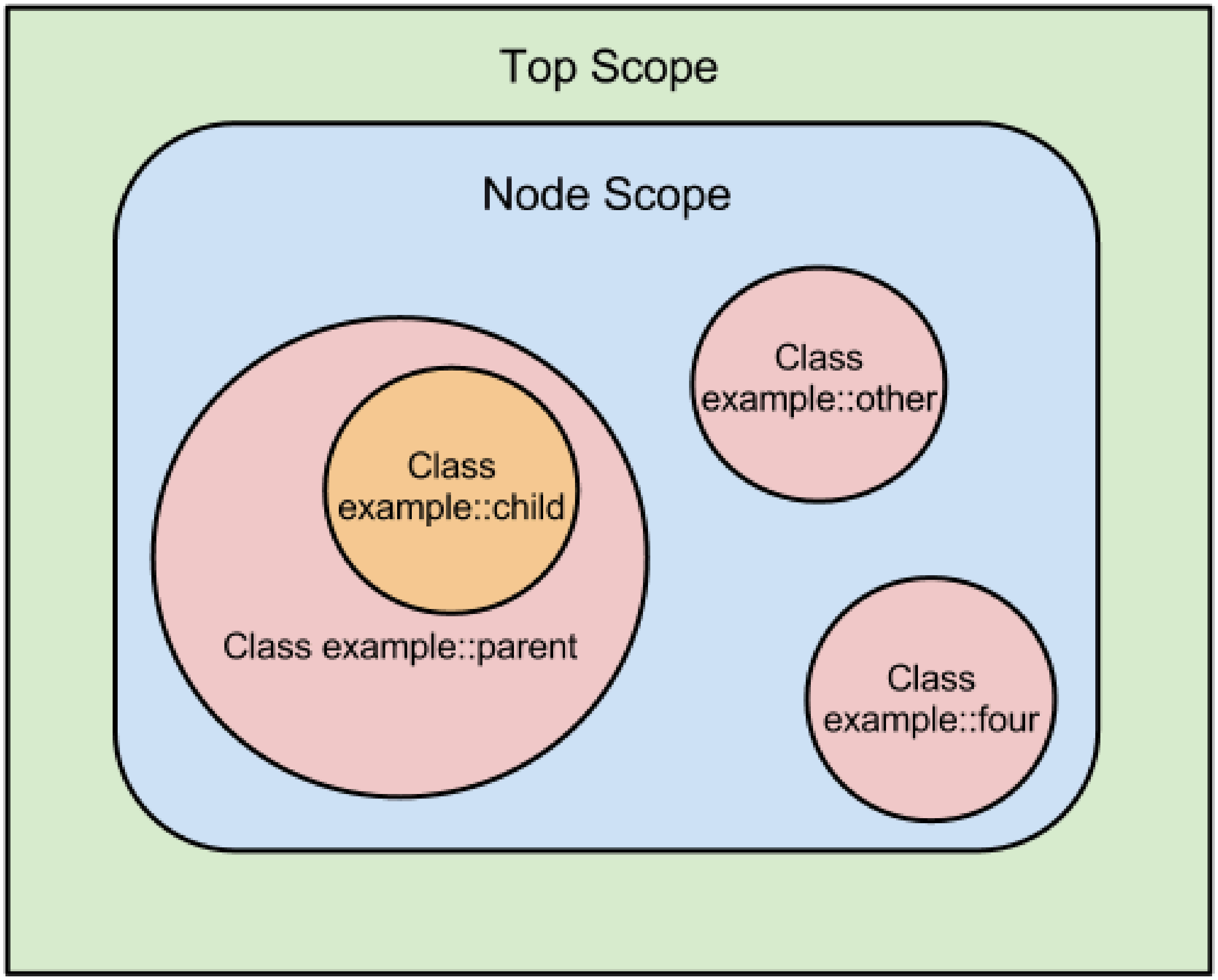
For more information, see Roles and profiles in the Puppet Enterprise documentation.

## scope

(or *variable scope*; includes *local scope*, *node scope*, and *top scope*)

The *scope* refers to an area of Puppet code that is partially isolated from other areas of code. Scopes limit the reach of *variables* and resource defaults. Scopes can be named (such as scopes created by class definitions) or anonymous (such as scopes created by *lambdas* and *defined resources*).

In general, if multiple variables with the same name are available, Puppet uses the variable from the most local scope.




The three most common scopes you'll work with in the Puppet language are:

- *Top scope*, from which variables are accessible from anywhere by their short name (such as `$my_variable`) but can be overridden in a local scope. The top scope's name is always an empty string, and top-scope variables can always be accessed using the double-colon namespace separator with an empty string (`$::my_variable`).
- *Node scope*, a special scope created in a *node definition*. You can refer to a node-scope variable from anywhere within a node scope by its short name, and node-scope variables can override top-scope variables.
- *Local scope*, where you can refer to a variable by its short name inside that scope and the scope's children, but typically cannot access it from other scopes.

There are a few exceptions, and you might also encounter other, more situational scopes. For detailed information, see Scope.

**Note:** Previously, Puppet used dynamic scope, which would search for short-named variables through a long chain of parent scopes. This scope was deprecated in Puppet 2.7, and in now is used only for resource defaults.

## singleton

A *singleton* is an object in the *Puppet language*, such as a *class*, that can only be evaluated one time. For example, you can't have more than one distinct class with the same specific *name* in a *manifest* or *catalog*, making that class a singleton.

## site

A *site* refers to an entire IT ecosystem that is managed by Puppet. A site includes all *master* servers, *agent nodes*, and independent *masterless* Puppet *nodes* within an organization.

## site module

A *site module* is a *module* that contains *classes* specific to a given Puppet *site*. These classes usually describe complete configurations for a specific system or group of systems. For example, the `site::db_replica` class might describe the entire configuration of a database server, and a new database server could be configured by applying that class to it.

## subclass

A *subclass* is a *class* that inherits from another class. See inheritance.

## subscribe

A notification *relationship* set with the `subscribe` *metaparameter*, or the wavy chaining arrow `~>`, is referred to as a *subscription*. For more information, see Relationships and ordering.

## template

A *template* is a partial document that is filled in with data from *variables*. Puppet can use Embedded Puppet (EPP) written in the *Puppet language*, or Embedded Ruby (ERB) templates written in Ruby, to generate configuration files tailored to an individual system. For more information, see Using templates.

## title

A *title* is the unique identifier of a *resource* or *class* in a given Puppet *catalog*.

- In a class, the title is simply the class's *name*.
- In a resource *declaration*, the title is the part after the first curly brace and before the colon; in the example below, the title is `/etc/passwd`:

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

- In native resource types, the name or *namevar* uses the title as its default *value* if you don't explicitly specify a name.
- In a *defined type* or a class, the title is available for use throughout the *definition* as the `$title` *variable*.

Unlike the name or namevar, a resource's title doesn't need to map to any *attribute* of the target system; it is only a referent. You can give a resource a single title even if its name must vary across different kinds of systems, like a configuration file whose location differs on Solaris.

For more information on resource titles, see Resources.

## top scope

See scope.

## type

A *type* is a kind of *resource* that Puppet is able to manage; for example, `file`, `cron`, and `service` are all resource types. A type specifies the set of attributes that a resource of that type can use, and models the behavior of that kind of resource on the target system. You can declare many resources of a given type.

Puppet ships with a set of built-in *resource types*; see the type reference for a complete list of them. New *native types* can be added as *plugins*, and *defined types* can be constructed by grouping together resources of existing types.

Contrast with type (data). See also type (defined) and type (native).

## type (data)

(or *data type*; includes *abstract data type* and *core data type*)

Every *value* has a *data type*, which is a named classification of a type of data that a *variable* or *parameter* can hold. The *Puppet language* has core data types (such as integer, Boolean, or string) and abstract data types (such as any or optional). For more information, see Values and data types.

## type (defined)

(or *defined type*, or *defined resource type*; sometimes called a *define* or *definition*)

A *defined type* is a *resource type* that is *defined* as a group of other *resources*, written in the *Puppet language*, and saved in a *manifest*. For example, a defined type could use a combination of `file` and `exec` resources to configure and populate a git repository.

After you define a type, new resources of that type can be *declared* just like any native or custom resource; these are called *defined resources*.

Because defined types are written in the Puppet language instead of as Ruby plugins, they are analogous to macros in other languages. Contrast with *native types*.

For more information, see Defined resource types .

## type (native)

(or *native type*, or *native resource type*)

A *native type* is a *resource* type that is written in Ruby. Puppet ships with a set of built-in native types, and custom native types can be distributed as *plugins* in *modules*. For a complete list of built-in types, see Resource types.

Native types have lower-level access to the target system than *defined types*, and can use the system's own tools to make changes. Most native types have one or more *providers* that can implement the same resources on different kinds of systems.

## value

In the *Puppet language*, a *value* is a piece of data which has a certain *data type*, or in some cases represents a literal data type. You can assign values to variables and parameters. For more information, see Values and data types.

## variable

A *variable* is a named placeholder in a *manifest* that represents a *value*. After they are assigned, variables cannot be reassigned within the same *scope*; however, other scopes might be able to assign their own value to any variable name.

Variables in Puppet are indicated with a dollar sign (`$operatingsystem`, also known as a short name) and assigned with the equals sign (`$operatingsystem = "Debian"`). In certain scopes, variables can also be accessed using a qualified name consisting of the scope name, followed by a double-colon namespace separator, then the variable name; this pattern can be repeated to drill down through multiple scopes. For example, `$apache::params::confdir` represents the `confdir` variable in the `params` subclass of the `apache` class.

*Facts* from *agents* are represented as variables within Puppet manifests, and are automatically pre-assigned before *compilation* begins. There are also several other pre-assigned variables; for more information, see Facts and built-in variables.

### variable scoping

See scope.

### virtual resource

A *virtual resource* is a resource that is declared in the *catalog* but isn't applied to a system unless it is explicitly *realized*.

For more information, see Virtual resources.

# Release notes

These release notes contain important information about Puppet® 6.7 and the Puppet agent.

This release incorporates new features, enhancements, and resolved issues from all previous releases. If you're upgrading from an earlier version of Puppet, check the release notes for any interim versions for details about additional improvements in this release over your current release.

Version numbers for Puppet and the agent use the format X.Y.Z, where:

- X must increase for major, backward-incompatible changes.
- Y can increase for backward-compatible new functionality or significant bug fixes.
- Z can increase for bug fixes.

Refer to these release notes for information about the following separate components, keeping in mind what version of each you have installed:

| | | |
|---|---|---|
| Puppet Server | PuppetDB | Facter |



## Puppet release notes

These are the new features, resolved issues, and deprecations in this version of Puppet.

### Puppet 6.7.2

Released 26 July 2019

#### Resolved issues

##### Catalog application failed with `puppetlabs-ciscopuppet` module

Catalog application failed when using a custom resource type whose automatic relationship method, such as `autorequire` or `autobefore`, returned an instance of `Puppet::Type` instead of a resource name. This occurred in particular when using the `puppetlabs-ciscopuppet` module. This issue was caused by a regression in Puppet 6.7.0. PUP-9926

#### Use of the `mailalias` resource type returned errors

This release fixes an issue affecting the `mailalias` resource type. Puppet was unable to correctly parse files, resulting in errors such as:

```
Could not prefetch mailalias provider 'aliases': Could not
parse line "mailer-daemon: postmaster" (file: /etc/aliases, line: 12
```

This issue was caused by a regression introduced in Puppet 6.7.0. PUP-9914

## Puppet 6.7.1

This version of Puppet was never released.

## Puppet 6.7.0

Released 23 July 2019

### New features

#### New `ca_fingerprint` setting verifies the CA bundle download against a fingerprint

This release adds a `ca_fingerprint` setting to specify that newly provisioned agents should verify the CA certificate when it is initially downloaded. This provides a way to securely bootstrap new agents. To enable this verification, set `ca_fingerprint` to the SHA256 digest of the CA certificate, which can be calculated on the Puppet Server. To calculate this SHA, run the following command on the master, or on the CA server if you have specified a server other than the master:

```
openssl dgst -sha256 -r /etc/puppetlabs/puppet/ssl/certs/ca.pem | cut -f1 -
d' '
```

PUP-9638

### Resolved issues

#### Some commands could not be found

Some Puppet commands, such as `puppet-infra`, might not be found in the system PATH. This fix ensures that the relevant directory, `opt/puppetlabs/bin`, is available in the PATH. PA-2750

#### Custom MSI actions are logged

Custom MSI actions did not correctly log `STDERR` to the MSI log. PA-2691

### Deprecations

#### Fine grained control of file and environment timeouts deprecated

Fine grained control of file and environment timeouts is deprecated. Instead, use `0` or `unlimited` to control default caching behavior and the environment-cache endpoint in Puppet Server's administrativeAPI to expire the cache as needed. PUP-9497

#### `SublocatedExpression` class

The AST `SublocatedExpression` class is no longer generated by the parser. The `SublocatedExpression`class itself will be removed from Puppet in a future release. PUP-9303

### Certificate authority subcommands and v1 CA HTTP API

Certificate authority subcommands have been removed from Puppet, including: `cert`, `ca`, `certificate`, `certificate request`, and `certificate_revocation_list`. Use `puppetserver ca` and `puppet ssl` instead. PUP-8998

As a part of the larger CA rework, the v1 CA HTTP API is removed (everything under the ca url /v1). PUP-3650

For details on changes and the new commands, see our documentation about certificates and SSL.

### Ruby certificate authority

Puppet no longer has a Ruby CA. All CA actions now rely entirely on the Clojure implementation in Puppet Server. It can be interacted with by means of the CA API and the `puppetserver ca` command, which leverages the API using subcommands like those provided by `puppet cert`. PUP-8912

### Trusted server facts

Trusted server facts are always enabled and have been deprecated since 5.0. This removes the setting and conditional logic. PUP-8530

### `write_only_yaml` node terminus

The `write_only_yaml` node terminus was used to "determine the list of nodes that the master knows about" and predated widespread PuppetDB adoption. The `write_only_yaml` has been deprecated since 4.10.5, and this commit removes it. Note this results in a Puppet Server speedup as it no longer needs to serialize node data as YAML to disk during a compile. PUP-8528

### LDAP node terminus

The LDAP node terminus has been removed. PUP-7601

### `computer`, `macauthorization`, and `mcx` types and providers

The `computer`, `macauthorization`, and `mcx` types and providers have been moved to the `macdslocal_core` module. It is not repackaged into puppet-agent in the 6.0 series.

### Nagios types

The Nagios types no longer ship with Puppet, and are now available as the `puppetlabs/nagios_core` module from the Forge.

### Cisco network devices

The Cisco network device types no longer ship with Puppet. These types and providers have been deprecated in favor of the `puppetlabs/cisco_ios` module, which is available on the Forge. PUP-8575

### `:undef` in types and providers

In previous versions, values from manifests assigned to resource attributes that contained undef values nested in arrays and hashes would use the Ruby symbol `:undef` to represent those values. When using `puppet apply` types and providers would see those as `:undef` or as the string "undef" depending on the implementation of the type. When using a master, the same values were correctly handled. In this version, Ruby nil is used consistently for this. (Top level undef values are still encoded as empty string for backwards compatibility). PUP-9112

### `puppet module build` command

To reduce the amount of developer tooling installed on all agents, this version of puppet removes the `puppet module build` command. To continue building module packages for the Forge and other repositories, install Puppet Development Kit (PDK). PUP-8763

### `pcore_type` and `pcore_value`

The earlier experimental `-rich_data` format used the tags `pcore_type` and `pcore_value`, these are now shortened to `__ptype` and `__pvalue` respectively. If you are using this experimental feature and have stored serializations you need to change them or write them again with the updated version. PUP-8597

### Webrick

Webrick support (previously deprecated) has been removed. To run Puppet as a server you must use Puppet Server. PUP-8591)

### `puppet master` command

The `puppet master` command and its subcommands have been removed. Instead, use a `puppet-config` command. PE-24280

### `–strict` flag in puppet module

The `–strict` flag in `puppet module` has been removed. The default behavior remains intact, but the tool no longer accepts non-strict versioning (such as release candidates and beta versions). PUP-8558

### Select settings

The following settings have been removed:

- The previously deprecated `configtimeout` setting has been removed in favor of the `http_connect_timeout` and `http_read_timeout` setting. PUP-8534
- The unused `ignorecache` setting has been removed. PUP-8533
- The previously deprecated `pluginsync` setting has now been removed. The agent’s pluginsync behavior is controlled based on whether it is using a cached catalog or not. PUP-8532
- The deprecated `app_management` setting has now been removed. Previously, this setting was ignored, and always treated as though it was set to be on. PUP-8531
- The deprecated `ordering` setting has been removed, and catalogs now always have the ordering previously provided by the `manifest` value of this setting. PUP-6165
- Settings related to the rack webserver from Puppet, including `binaddress` and `masterhttplog`. PUP-3658

### String duplication in 3x runtime converter

Types and provider implementations must not mutate the parameter values of a resource. With this release, it is more likely that the parameters of a resource have frozen (that is, immutable) string values and any type or provider that directly mutates a resource parameter might fail. Previously, every resource attribute was copied to not make application break even if they did mutate. Look for use of `gsub!` in your modules and replace logic with non-mutating version, or operate on a copy of the value. All authors of Forge modules having this problem have been notified. PUP-7141

### `Puppet.newtype` method

The deprecated `Puppet.newtype` method (deprecated since 2011) has now been removed. ( PUP-7078)

### Certificate handling commands deprecated but not removed

The following subcommands were deprecated in a previous version and slated for removal in this version. While these subcommands are still deprecated, they have not yet been removed.

- `ca_name`
- `cadir`
- `cacert`
- `cakey`

- `capub`
- `cacrl`
- `caprivatedir`
- `csrdir`
- `signeddir`
- `capass`
- `serial`
- `autosign`
- `allow_duplicate_certs`
- `ca_ttl`
- `cert_inventory`

## Puppet known issues

These are the known issues in this version of Puppet.

### User and group management on macOS 10.14 requires Full Disk Access

To manage users and groups with Puppet on macOS 10.14, you must grant Puppet Full Disk Access (FDA). You must also grant FDA to the parent process that triggers your Puppet run. For example:

- To run Puppet in a master-agent infrastructure, you must grant FDA to the `pxp-agent`.
- To run Puppet from a remote machine with SSH commands, you must grant FDA to `sshd`.
- To run Puppet commands from the terminal, you must grant FDA to `terminal.app`.

To give Puppet access on a machine running macOS10.14, go to **System Preferences** > **Security & Privacy** > **Privacy** > **Full Disk Access**, and add the path to the Puppet executable, along with any other parent processes you use to run. Alternatively, set up automatic access using Privacy Preferences Control Profiles and a Mobile Device Management Server. PA-2226, PA-2227

# Managing essential configuration with Puppet

Try out some common configuration tasks to see how you can use Puppet to manage your IT infrastructure. These instructions assume you have installed Puppet and at least one agent, and that you work through them in the order presented.



## Get started managing an NTP service

Network Time Protocol (NTP) is one of the most crucial, yet easiest, services to configure and manage with Puppet, to properly synchronize time across all your nodes. Follow this guide to get started managing a NTP service using the Puppet `ntp` module.

Ensure you've already installed , and at least one *nix agent. Also, log in as root or Administrator on your nodes.

The clocks on your servers are not inherently accurate. They need to synchronize with something to let them know what the right time is. NTP is a protocol that synchronizes the clocks of computers over a network. NTP uses Coordinated Universal Time (UTC) to synchronize computer clock times to within a millisecond.

Your entire datacenter, from the network to the applications, depends on accurate time for security services, certificate validation, and file sharing across Puppet agents. If the time is wrong, your Puppet master might mistakenly issue agent certificates from the distant past or future, which other agents treat as expired.

Using the Puppet NTP module, you can:

- Ensure time is correctly synced across all the servers in your infrastructure.
- Ensure time is correctly synced across your configuration management tools.
- Roll out updates quickly if you need to change or specify your own internal NTP server pool.

This guide walks you through the following steps in setting up NTP configuration management:

- Installing the `puppetlabs-ntp` module.
- Adding classes to the `default` node in your main manifest.
- Viewing the status of your NTP service.
- Using multiple nodes in the main manifest to configure NTP for different permissions.

**Note:** You can add the NTP service to as many agents as needed. For simplicity, this guide describes adding it to only one.

1. The first step is installing the `puppetlabs-ntp` module. The `puppetlabs-ntp` module is part of the supported modules program; these modules are supported, tested, and maintained by Puppet. For more information on `puppetlabs-ntp`, see the README. To install it, run:

   ```
   puppet module install puppetlabs-ntp
   ```

   The resulting output is similar to this:

   ```
    Preparing to install into /etc/puppetlabs/puppet/modules ...
       Notice: Downloading from http://forgeapi.puppetlabs.com ...
       Notice: Installing -- do not interrupt ...
       /etc/puppetlabs/puppet/environments/production/modules
       └── puppetlabs-ntp (v3.1.2)
   ```

   That's it! You've just installed the `puppetlabs-ntp` module.
2. The next step is adding classes from the NTP module to the main manifest.

   The NTP module contains several classes. Classes are named chunks of Puppet code and are the primary means by which Puppet configures nodes. The NTP module contains the following classes:

   - `ntp`: the main class, which includes all other NTP classes, including the classes in this list.
   - `ntp::install`: handles the installation packages.
   - `ntp::config`: handles the configuration file.
   - `ntp::service`: handles the service.

   You're going to add the `ntp` class to the default node in your main manifest. Depending on your needs or infrastructure, you might have a different group that you'll assign NTP to, but you would take similar steps.

   a) From the command line on the master, navigate to the directory that contains the main manifest:

      ```
      cd /etc/puppetlabs/code/environments/production/manifests
      ```

   b) Use your text editor to open `site.pp`.
   c) Add the following Puppet code to `site.pp`:

      ```
      node default {
        class { 'ntp':
              servers => ['nist-time-server.eoni.com','nist1-
      lv.ustiming.org','ntp-nist.ldsbc.edu']
        }
      }
      ```

      **Note:** If your `site.pp` file already has a default node in it, add just the `class` and `servers` lines to it.

      **Note:** For additional time server options, see the list at https://www.ntppool.org/.
   d) On your agent, start a Puppet run:

      ```
      puppet agent -t
      ```

      Your Puppet-managed node is now configured to use NTP.
3. To check if the NTP service is running, run:

   ```
   puppet resource service ntpd
   ```

   The result looks like this:

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

**4.** If you want to configure the NTP service to run differently on different nodes, you can set up NTP on nodes other than `default` in the `site.pp` file.

In previous steps, you've been configuring the default node.

In the example below, two NTP servers (`kermit` and `grover`) are configured to talk to outside time servers. The other NTP servers (`snuffie`, `bigbird`, and `hooper`) use those two primary servers to sync their time.

One of the primary ntp servers, `kermit`, is very cautiously configured — it can't afford outages, so it's not allowed to automatically update its NTP server package without testing. The other servers are more permissively configured.

The `site.pp` looks like this:

```
node "kermit.example.com" {
  class { "ntp":
        servers    => [ '0.us.pool.ntp.org iburst','1.us.pool.ntp.org
 iburst','2.us.pool.ntp.org iburst','3.us.pool.ntp.org iburst'],
        autoupdate => false,
        restrict   => [],
        enable     => true,
  }
}

node "grover.example.com" {
  class { "ntp":
        servers    => [ 'kermit.example.com','0.us.pool.ntp.org
 iburst','1.us.pool.ntp.org iburst','2.us.pool.ntp.org iburst'],
        autoupdate => true,
        restrict   => [],
        enable     => true,
  }
}

node "snuffie.example.com", "bigbird.example.com", "hooper.example.com" {
  class { "ntp":
        servers    => [ 'grover.example.com', 'kermit.example.com'],
        autoupdate => true,
        enable     => true,
  }
}
```

In this way, it is possible to configure NTP on multiple nodes to suit your needs.

For more information about working with the `puppetlabs-ntp` module, check out our How to Manage NTP webinar.

offers many opportunities for learning and training, from formal certification courses to guided online lessons. See the Learning page for more information.

## Get started managing a DNS nameserver file

A nameserver ensures that the human-readable URLs you type in your browser (for example, `example.com`) resolve to IP addresses that computers can read. This guide helps you get started managing a simple Domain Name System (DNS) nameserver file with Puppet.

Before starting this walk-through, complete the previous exercise of setting up NTP management. Log in as root or Administrator on your nodes.

Sysadmins typically need to manage a nameserver file for internal resources that aren't published in public nameservers. For example, suppose you have several employee-maintained servers in your infrastructure, and the DNS network assigned to those servers use Google's public nameserver located at `8.8.8.8`. However, there are several resources behind your company's firewall that your employees need to access on a regular basis. In this case, you'd build a private nameserver (for example at `10.16.22.10`), and use Puppet to ensure all the servers in your infrastructure have access to it.

In this exercise, you learn how to:

- Write a module that contains a class called `resolver` to manage a nameserver file called `/etc/resolv.conf`.
- Enforce the desired state of that class from the command line of your Puppet agent.

**Note:** You can add the DNS nameserver class to as many agents as needed. For simplicity, this guide describes adding it to only one.

**1.** The first step is creating the `resolver` module and a template.

While some modules are large and complex, this module module contains just one class and one template

By default, Puppet keeps modules in an environment's modulepath, which for the production environment defaults to `/etc/puppetlabs/code/environments/production/modules`. This directory contains modules that Puppet installs, those that you download from the Forge, and those you write yourself.

**Note:** Puppet creates another module directory: `/opt/puppetlabs/puppet/modules`. Don't modify or add anything in this directory.

For thorough information about creating and using modules, see Modules fundamentals, the Beginner's guide to modules, and the Puppet Forge .

Modules are directory trees. For this task, you'll create a directory for the `resolver` module, a subdirectory for its templates, and a template file that Puppet uses to create the `/etc/resolv.conf` file that manages DNS.

a) From the command line on the Puppet master, navigate to the modules directory:

```
cd /etc/puppetlabs/code/environments/production/modules
```

b) Create the module directory and its templates directory:

```
mkdir -p resolver/templates
```

c) Use your text editor to create a file called `resolv.conf.erb` inside the `resolver/templates` directory.

d) Edit the `resolv.conf.erb` file to add the following Ruby code:

```
# Resolv.conf generated by Puppet

<% [@nameservers].flatten.each do |ns| -%>
nameserver <%= ns %>
<% end -%>
```

This Ruby code is a template for populating `/etc/resolv.conf` correctly, no matter what changes are manually made to `/etc/resolv.conf`, as you see in a later step.

e) Save and exit the file.

That's it! You've created a Ruby template to populate `/etc/resolv.conf`.

**2.** Add managing aaaathe `resolv.conf` file to your main manifest.

a) On the master, open `/etc/resolv.conf` with your text editor, and copy the IP address of your master's nameserver. In this example, the nameserver is `10.0.2.3`.

b) Navigate to the main manifest:

```
cd /etc/puppetlabs/code/environments/production/manifests
```

c) Use your text editor to open the `site.pp` file and add the following Puppet code to the default node, making the `nameservers` value match the one you found in `/etc/resolv.conf`:

```
$nameservers = ['10.0.2.3']

file { '/etc/resolv.conf':
  ensure  => file,
  owner   => 'root',
  group   => 'root',
  mode    => '0644',
  content => template('resolver/resolv.conf.erb'),
}
```

d) From the command line on your agent, run Puppet: `puppet agent -t`

To see the results in the resolve.conf file, run:

```
cat /etc/resolv.conf
```

The file contains the nameserver you added to your main manifest.

That's it! You've written and applied a module that contains a class that ensures your agents resolve to your internal nameserver.

Note the following about your new class:

- It ensures the creation of the file `/etc/resolv.conf`.
- The content of `/etc/resolv.conf` is modified and managed by the template, `resolv.conf.erb`.

**3.** Finally, let's take a look at how Puppet ensures the desired state of the `resolver` class on your agents. In the previous task, you set the nameserver IP address. Now, simulate a scenario where a member of your team changes the contents of `/etc/resolv.conf` to use a different nameserver and, as a result, can no longer access any internal resources:

a) On the agent to which you applied the `resolver` class, edit `/etc/resolv.conf` to contain any nameserver IP address other than the one you want to use.

b) Save and exit the file.

c) Now, fix the mistake you've introduced. From the command line on your agent, run: `puppet agent -t --onetime`

To see the resulting contents of the managed file, run:

```
cat /etc/resolv.conf
```

Puppet has enforced the desired state of the agent node by changing the `nameserver` value back to what you specified in `site.pp` on the master.

For more information about working with Puppet and DNS, see our Dealing with Name Resolution Issues blog post.

offers many opportunities for learning and training, from formal certification courses to guided online lessons. See the Learning page for more information.

## Get started managing sudo privileges

Managing sudo on your agents allows you to control which system users have access to elevated privileges. This guide helps you get started managing sudo privileges across your nodes, using a module from the Puppet Forge in conjunction with a simple module you write.

Before starting this walk-through, complete the previous exercises.

Ensure you've already installed , and at least one *nix agent. Also, log in as root or Administrator on your nodes.

Using this guide, you learn how to:

- Install the `saz-sudo` module as the foundation for managing sudo privileges.
- Write a module that contains a class called `privileges` to manage a resource that sets privileges for certain users.
- Add classes from the `privileges` and `sudo` modules to your agents.

**Note:** You can add the `sudo` and `privileges` classes to as many agents as needed. For simplicity, this guide describes only one.

1. Start by installing the `saz-sudo` module. It's available on the Forge, and is one of many modules written by a member of the Puppet user community. You can learn more about the module at forge.puppet.com/saz/sudo. To install the `saz-sudo` module, run the following command on the master:

   ```
   puppet module install saz-sudo
   ```

   The resulting output is similar to this:

   ```
   Preparing to install into /etc/puppetlabs/code/environments/production/
   modules …
   Notice: Downloading from http://forgeapi.puppetlabs.com ...
       Notice: Installing -- do not interrupt ...
       /etc/puppetlabs/puppet/modules
       └── saz-sudo (v2.3.6)
             └── puppetlabs-stdlib (3.2.2) [/opt/puppet/share/puppet/modules]
   ```

   That's it! You've installed the `saz-sudo` module.
2. Next, you'll create a module that contains the `privileges` class.

   Like in the DNS exercise, this is a small module with just one class. You'll create the `privileges` module directory, its `manifests` subdirectory, and an `init.pp` manifest file that contains the `privileges` class.

   a) From the command line on the master, navigate to the modules directory:

   ```
   cd /etc/puppetlabs/code/environments/production/modules
   ```

   b) Create the module directory and its manifests directory:

   ```
   mkdir -p privileges/manifests
   ```

   c) In the manifests directory, use your text editor to create the `init.pp` file, and edit it so it contains the following Puppet code:

   ```
   class privileges {

      sudo::conf { 'admins':
      ensure  => present,
      content => '%admin ALL=(ALL) ALL',
      }
   ```

```
 }
```

The `sudo::conf 'admins'` line creates a sudoers rule that ensures that members of the `admins` group have the ability to run any command using sudo. This resource creates a configuration fragment file to define this rule in `/etc/sudoers.d/`. It's called something like `10_admins`.

d) Save and exit the file.

That's it! You've created a module that contains a class that, after it's applied, ensures that your agents have the correct sudo privileges set for the root user and the `admins` and wheel groups.

**3.** Next, add the `privileges` and `sudo` classes to default nodes.

a) From the command line on the master, navigate to the main manifest:

```
cd /etc/puppetlabs/code/environments/production/manifests
```

b) Open `site.pp` with your text editor and add the following Puppet code to the default node:

```
class { 'sudo': }
sudo::conf { 'web':
  content  => "web ALL=(ALL) NOPASSWD: ALL",
}
class { 'privileges': }
sudo::conf { 'jargyle':
  priority => 60,
  content  => "jargyle ALL=(ALL) NOPASSWD: ALL",
}
```

The `sdo::conf ‘web’` line creates a sudoers rule to ensure that members of the `web` group can run any command using sudo. This resource creates a configuration fragment file to define this rule in `/etc/sudoers.d/`.

The `sudo::conf ‘jargyle’` line creates a sudoers rule to ensure that the user `jargyle` can run any command using sudo. This resource creates a configuration fragment to define this rule in `/etc/sudoers.d/`. It's called something like `60_jargyle`.

c) Save and exit the file.

d) On your master, ensure that there are no errors:

```
puppet parser validate site.pp
```

The parser returns nothing if there are no errors.

e) From the command line on your agent, run Puppet: `puppet agent -t`

That's it! You have successfully applied `sudo` and `privileges` classes to nodes.

f) To confirm it worked, run the following command on an agent:

```
sudo -l -U jargyle
```

The results should resemble the following:

```
 Matching Defaults entries for jargyle on this host:
!visiblepw, always_set_home, env_reset, env_keep="COLORS DISPLAY
 HOSTNAME HISTSIZE
INPUTRC KDEDIR LS_COLORS", env_keep+="MAIL PS1 PS2 QTDIR USERNAME LANG
 LC_ADDRESS
LC_CTYPE", env_keep+="LC_COLLATE LC_IDENTIFICATION LC_MEASUREMENT
 LC_MESSAGES",
env_keep+="LC_MONETARY LC_NAME LC_NUMERIC LC_PAPER LC_TELEPHONE",
 env_keep+="LC_TIME
LC_ALL LANGUAGE LINGUAS _XKB_CHARSET XAUTHORITY",
```

```
secure_path=/usr/local/bin\:/sbin\:/bin\:/usr/sbin\:/usr/bin

 User jargyle may run the following commands on this host:
(ALL) NOPASSWD: ALL
```

For more information about working with Puppet and sudo users, see our Module of The Week: saz/sudo - Manage sudo configuration blog post.

offers many opportunities for learning and training, from formal certification courses to guided online lessons. See the Learning page for more information.

## Get started managing firewall rules

With a firewall, admins define firewall rules, which sets a policy for things like application ports (TCP/UDP), network ports, IP addresses, and accept-deny statements. This guide helps you get started managing firewall rules with Puppet.

Before starting this walk-through, complete the previous exercises in the common configuration tasks.

Ensure you've already installed , and at least one *nix agent. Also, log in as root or Administrator on your nodes.

Firewall rules are applied with a top-to-bottom approach. For example, when a service, say SSH, attempts to access resources on the other side of a firewall, the firewall applies a list of rules to determine if or how SSH communications are handled. If a rule allowing SSH access can't be found, the firewall denies access to that SSH attempt.

To best way to manage firewall rules with Puppet is to divide them into `pre` and `post` groups to ensure Puppet checks them in the correct order.

Using this guide, you learn how to:

- Install the `puppetlabs-firewall` module.
- Write a module to define the firewall rules for your Puppet managed infrastructure.
- Add the firewall module to the main manifest.
- Enforce the desired state using the `my_firewall` class.

1. The first step is installing the `puppetlabs-firewall` module from the Puppet Forge. The module introduces the `firewall` resource, which is used to manage and configure firewall rules. For more information about the `puppetlabs-firewall` module, see its README. To install the module, on the master, run:

   ```
   puppet module install puppetlabs-firewall
   ```

   The resulting output is similar to this:

   ```
   Preparing to install into /etc/puppetlabs/puppet/environments/production/
   modules ...
       Notice: Downloading from https://forgeapi.puppetlabs.com ...
       Notice: Installing -- do not interrupt ...
       /etc/puppetlabs/puppet/environments/production/modules
       └── puppetlabs-firewall (v1.6.0)
   ```

   That's it! You've just installed the `firewall` module.

**2.** Next, you'll write the `my_firewall` module, which contains three classes. You'll create the `my_firewall` module directory, its `manifests` subdirectory, a `pre.pp` manifest file and a `post.pp` manifest file.

a) From the command line on the master, navigate to the modules directory:

```
cd /etc/puppetlabs/code/environments/production/modules
```

b) Create the module directory and its manifests directory:

```
mkdir -p my_firewall/manifests
```

c) From the `manifests` directory, use your text editor to create `pre.pp`.

d) The pre rules are rules that the firewall applies when a service requests access. It is run before any other rules. Edit `pre.pp` so it contains the following Puppet code:

```
class my_firewall::pre {
  Firewall {
    require => undef,
  }
     firewall { '000 accept all icmp':
       proto  => 'icmp',
       action => 'accept',
     }
     firewall { '001 accept all to lo interface':
       proto   => 'all',
       iniface => 'lo',
       action  => 'accept',
     }
     firewall { '002 reject local traffic not on loopback interface':
       iniface     => '! lo',
       proto       => 'all',
       destination => '127.0.0.1/8',
       action      => 'reject',
     }
     firewall { '003 accept related established rules':
       proto  => 'all',
       state  => ['RELATED', 'ESTABLISHED'],
       action => 'accept',
     }
   }
```

These default rules allow basic networking to ensure that existing connections are not closed.

e) Save and exit the file.

f) From the `manifests` directory, use your text editor to create `post.pp`.

g) The post rules tell the firewall to drop requests that haven't met the rules defined by `pre.pp` or in `site.pp`. Edit `post.pp` so it contains the following Puppet code:

```
 class my_firewall::post {
     firewall { '999 drop all':
       proto  => 'all',
       action => 'drop',
       before => undef,
     }
   }
```

h) Save and exit the file.

That's it! You've written a module that contains a class that, after it's applied, ensures your firewall has rules in it that are managed by Puppet.

**3.** Now you'll add the `firewall` module to the main manifest so that Puppet is managing firewall configuration on nodes.

a) On the master, navigate to the main manifest:

```
cd /etc/puppetlabs/code/environments/production/manifests
```

b) Use your text editor to open `site.pp`.

c) Add the following Puppet code to your `site.pp` file:

```
  resources { 'firewall':
     purge => true,
   }
```

This clears any existing rules and make sure that only rules defined in Puppet exist on the machine.

d) Add the following Puppet code to your `site.pp` file:

```
 Firewall {
     before  => Class['my_firewall::post'],
     require => Class['my_firewall::pre'],
   }

   class { ['my_firewall::pre', 'my_firewall::post']: }
```

These settings ensure that the pre and post classes are run in the correct order to avoid locking you out of your node during the first Puppet run, and declaring `my_firewall::pre` and `my_firewall::post` satisfies the specified dependencies.

e) Add the `firewall` class to your `site.pp` to ensure the correct packages are installed:

```
class { 'firewall': }
```

f) To apply the configuration, on the agent, run Puppet: `puppet agent -t`

That's it! Puppet is now managing the firewall configuration on the agent.

g) To check your firewall configuration, on the agent, run: `iptables --list`

The resulting output is similar to this:

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     icmp --  anywhere             anywhere            /* 000
 accept all icmp */
ACCEPT     all  --  anywhere             anywhere            /* 001
 accept all to lo interface */
REJECT     all  --  anywhere             loopback/8          /* 002
 reject local traffic not on loopback interface */ reject-with icmp-
port-unreachable
ACCEPT     all  --  anywhere             anywhere            /* 003
 accept related established rules */ state RELATED,ESTABLISHED
DROP       all  --  anywhere             anywhere            /* 999 drop
 all */

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

4. Finally, let's take a look at how Puppet ensures the desired state of the `my_firewall` class on your agents. In the previous step, you applied the firewall rules. Now, simulate a scenario where a member of your team changes the contents the `iptables` to allow connections on a random port that was not specified in `my_firewall`:
   a) On an agent where you applied the `my_firewall` class, run:

   ```
   iptables --list
   ```

   Note that the rules from the `my_firewall` class have been applied.
   b) From the command line, add a rule to allow connections to port 8449 by running:

   ```
   iptables -I INPUT -m state --state NEW -m tcp -p tcp --dport 8449 -j
    ACCEPT
   ```
   c) Run `iptables --list` again and see that this new rule is now listed.
   d) Run Puppet on that agent:

   ```
   puppet agent -t --onetime
   ```
   e) Run `iptables --list` on that node one more time, and notice that Puppet has enforced the desired state you specified for the firewall rules

   That's it! Puppet has enforced the desired state of your agent.

You can learn more about the Puppet firewall module by visiting the Forge .

offers many opportunities for learning and training, from formal certification courses to guided online lessons. See the Learning page for more information.

# Installing and upgrading

Puppet can be installed and upgraded in various configurations to fit the needs of your environment.

- System requirements on page 45
Puppet system requirements depend on your deployment type and size. Before installing, ensure your systems are compatible with infrastructure and agent requirements.
- Using the Puppet platform on page 47
The Puppet platform bundles the components needed for a successful deployment.
- Installing Puppet on page 51
For the agent-master architecture, you must install Puppet Server and, optionally, PuppetDB.
- Installing agents on page 51
You can install agents on *nix, Windows, or macOS.
- Upgrading on page 57
To upgrade your deployment, you must upgrade both the infrastructure components and agents.

## System requirements

Puppet system requirements depend on your deployment type and size. Before installing, ensure your systems are compatible with infrastructure and agent requirements.

**Note:** Puppet masters must run on some variant of *nix. You can't run masters on Windows.

## Hardware requirements

The agent service has no particular hardware requirements and can run on nearly anything. The master, however, is fairly resource intensive, and must be installed on a robust, dedicated server.

The demands on the master vary widely between deployments. Resource needs are affected by the number of agents being served, how frequently agents check in, how many resources are being managed on each agent, and the complexity of the manifests and modules in use.

These minimum hardware requirements are based on internal testing.

| Node volume | Cores | Heap | `ReservedCodeCache` |
|---|---|---|---|
| dozens | 2 | 1 GB | n/a |
| 1,000 | 2-4 | 4 GB | 512m |

## Supported agent platforms

Puppet provides official packages for various operating systems and versions. You aren't necessarily limited to using official packages, but installation and maintenance is generally easier with official tested packages.

### Packaged platforms

`puppet-agent` packages are available for these platforms.

Packages for tested versions are officially tested. Packages for untested versions might not be automatically tested.

| Operating system | Tested versions | Untested versions |
|---|---|---|
| Debian | 8, 9 | |
| Fedora | 28, 29, 30 | |
| macOS | 10.12 Sierra, 10.13 High Sierra, 10.14 Mojave (see note below) | |
| Microsoft Windows | 10 Enterprise | 7, 8, 10 |
| Microsoft Windows Server | 2008R2, 2012R2, 2016, 2019 | 2008, 2012 |
| Red Hat Enterprise Linux, including:<br>• Amazon Linux v1 (using RHEL 6 packages)<br>• Amazon Linux v2 (using RHEL 7 packages) | 5, 6, 7 | 8 (beta) |
| SUSE Linux Enterprise Server | 15 | |
| Ubuntu | 14.04, 16.04 | |

**Note:** On macOS 10.14 Mojave, you must grant Puppet Full Disk Access to be able to manage users and groups. To give Puppet access on a machine running macOS 10.14, go to **System Preferences** > **Security & Privacy** > **Privacy** > **Full Disk Access**, and add the path to the Puppet executable. Alternatively, set up automatic access using Privacy Preferences Control Profiles and a Mobile Device Management Server.

### Dependencies

If you install using an official package, your system's package manager ensures that dependencies are installed. If you install the agent on a platform without a supported package, you must manually install these packages, libraries, and gems:

- Ruby 2.5.x
- CFPropertyList 2.2 or later

- Facter 2.0 or later
- The `msgpack` gem, if you're using msgpack serialization

## Timekeeping and name resolution

Before installing , there are network requirements you need to consider and prepare for. The most important requirements include syncing time and creating a plan for name resolution.

### Timekeeping

Use NTP or an equivalent service to ensure that time is in sync between your master, which acts as the certificate authority, and any agent nodes. If time drifts out of sync in your infrastructure, you might encounter issues such as agents recieving outdated certificates. A service like NTP (available as a supported module) ensures accurate timekeeping.

### Name resolution

Decide on a preferred name or set of names that agent nodes can use to contact the master. Ensure that the master can be reached by domain name lookup by all future agent nodes.

You can simplify configuration of agent nodes by using a CNAME record to make the master reachable at the hostname `puppet`, which is the default master hostname that is suggested when installing an agent node.

## Firewall configuration

In the agent-master architecture, your master must allow incoming connections on port 8140, and agent nodes must be able to connect to the master on that port.

# Using the Puppet platform

The Puppet platform bundles the components needed for a successful deployment.

### Puppet platform components

The Puppet platform includes these components:

| Package | Contents |
|---|---|
| `puppet-agent` | Puppet, Facter, Hiera, the PXP agent, root certificates, and prerequisites like Ruby and Augeas.<br><br>**Note:** In Puppet version 3.8 and later, Enterprise Linux 5 packages contain only the agent component. |
| `puppetserver` | Puppet Server |
| `puppetdb` | PuppetDB |
| `puppetdb-termini` | Plugins to connect your master to PuppetDB |

**Note:** The Puppet platform is available only for Linux, because most of its components can run only on Linux. The `puppet-agent` component is available independently for Windows and macOS.

## Puppet platform location and naming

The Puppet platform is packaged separately for each supported operating system and version.

*nix platform packages are located in a Puppet.com repository corresponding to the package management system.

| Package management system | URL naming convention | URL example |
|---|---|---|
| Yum | https://yum.puppet.com/<PLATFORM_NAME>-release-<OS ABBREVIATION>-<OS VERSION>.noarch.rpm | https://yum.puppet.com/puppet6-release-el-7.noarch.rpm |
| Apt | https://apt.puppet.com/<PLATFORM_VERSION>-release-<VERSION CODE NAME>.deb<br>**Tip:** For Ubuntu releases, the code name is the adjective, not the animal. | https://apt.puppet.com/puppet6-release-wheezy.deb |

Windows and macOS agent packages are indexed on the Puppet download site.

## Managing platform versions

To receive the most up-to-date software without introducing breaking changes, use the `latest` platform, pin your infrastructure to known versions, and update the pinned version manually when you're ready to update.

For example, if you're using the `puppetlabs/puppet_agent` module to manage the installed `puppet-agent` package, use this resource to pin it to version 6.0:

```
class { '::puppet_agent':
  collection      => 'latest',
  package_version => '6.0.0',
}
```

If you're upgrading from a 1.x version of `puppet-agent`, simply update the `package_version` when you're ready to upgrade to the 6.x series.

## Enable the Puppet platform repository

Enabling the Puppet platform repository makes the components needed for installation available on your system.

The process for enabling the repository varies based on your package management system.

### Enable the Puppet platform on Yum

Identify the URL of the package you want to enable based on your operating system and version. For details, see Puppet platform location and naming on page 47.

Logged in as root, run the RPM tool in upgrade mode:

```
sudo rpm -U <PACKAGE_URL>
```

**Note:** For Enterprise Linux 5, you must download the package before running RPM:

```
wget https://yum.puppet.com/puppet6-release-el-5.noarch.rpm
sudo rpm -Uvh puppet6-release-el-5.noarch.rpm
```

For example, to enable the Enterprise Linux 7 repository:

```
sudo rpm -Uvh https://yum.puppet.com/puppet6-release-el-7.noarch.rpm
```

### Enable the Puppet platform on Apt

Identify the URL of the package you want to enable based on your operating system and version. For details, see Puppet platform location and naming on page 47.

1. Logged in as root, download the package and run the dpkg tool in install mode:

   ```
   wget <PACKAGE_URL>
   sudo dpkg -i <FILE_NAME>.deb
   ```

   For example, to enable the Ubuntu 16.04 Xenial repository:

   ```
   wget https://apt.puppetlabs.com/puppet6-release-xenial.deb
   sudo dpkg -i puppet6-release-xenial.deb
   ```

2. Update the `apt` package lists: `sudo apt-get update`

## Verify packages

Puppet signs most of its packages, Ruby gems, and release tarballs with GNU Privacy Guard (GPG). This signature proves that the packages originate from Puppet and have not been compromised. Security-conscious users can use GPG to verify package signatures.

**Tip:**

Certain operating systems and installation methods automatically verify package signatures. In these cases, you don't need to do anything to verify the package signature.

- If you install from the Puppet Yum and Apt repositories, the release package that enables the repository also installs our release signing key. The Yum and Apt tools automatically verify the integrity of packages as you install them.
- If you install a Windows agent using an .msi package, the Windows installer automatically verifies the signature before installing the package.

### Verify a source tarball or gem

You can manually verify the signature for Puppet source tarballs or Ruby gems.

1. Import the public key: `gpg --keyserver pgp.mit.edu --recv-key 7F438280EF8D349F`

   The key is also available via HTTP.

   **Tip:** If this is your first time running the gpg tool, it might fail to import the key after creating its configuration file and keyring. You can run the command a second time to import the key into your newly created keyring.

   The gpg tool imports the key:

   ```
   gpg: requesting key EF8D349F from hkp server pgp.mit.edu gpg: /home/
   username/.gnupg/trustdb.gpg: trustdb created gpg: key EF8D349F:
    public key "Puppet, Inc. Release Key (Puppet, Inc. Release Key)
    <release@puppet.com>" imported gpg: no ultimately trusted keys found gpg:
    Total number processed: 1 gpg: imported: 1 (RSA: 1)
   ```

2. Verify the fingerprint: `gpg --list-key --fingerprint 7F438280EF8D349F`

   The fingerprint of the  release signing key is 6F6B 1550 9CF8 E59E 6E46 9F32 7F43 8280 EF8D 349F. Ensure the fingerprint listed matches this value.

3. Download the tarball or gem and its corresponding `.asc` file from https://downloads.puppetlabs.com/puppet/.
4. Verify the tarball or gem, replacing <VERSION> with the Puppet version number, and <FILE TYPE> with `tar.gz` for a tarball or `gem` for a Ruby gem: `gpg --verify puppet-<VERSION>.<FILE TYPE>.asc puppet-<VERSION>.<FILE TYPE>`

   The output confirms that the signature matches:

   ```
   gpg: Signature made Mon 19 Sep 2016 04:58:29 PM UTC using RSA key ID
    EF8D349F
   ```

```
gpg: Good signature from "Puppet, Inc. Release Key (Puppet, Inc. Release
 Key) <release@puppet.com>"
```

**Tip:** If you haven't set up a trust path to the key, you receive a warning that the key is not certified. If you've verified the fingerprint of the key, GPG has verified the archive's integrity; the warning simply means that GPG can't automatically prove the key's ownership.

### Verify an RPM package

RPM packages include an embedded signature, which you can verify after importing the Puppet public key.

1. Import the public key: `gpg --keyserver pgp.mit.edu --recv-key 7F438280EF8D349F`

   The key is also available via HTTP.

   **Tip:** If this is your first time running the gpg tool, it might fail to import the key after creating its configuration file and keyring. You can run the command a second time to import the key into your newly created keyring.

   The gpg tool imports the key:

   ```
   gpg: requesting key EF8D349F from hkp server pgp.mit.edu gpg: /home/
   username/.gnupg/trustdb.gpg: trustdb created gpg: key EF8D349F:
    public key "Puppet, Inc. Release Key (Puppet, Inc. Release Key)
    <release@puppet.com>" imported gpg: no ultimately trusted keys found gpg:
    Total number processed: 1 gpg: imported: 1 (RSA: 1)
   ```
2. Verify the fingerprint: `gpg --list-key --fingerprint 7F438280EF8D349F`

   The fingerprint of the release signing key is 6F6B 1550 9CF8 E59E 6E46 9F32 7F43 8280 EF8D 349F. Ensure the fingerprint listed matches this value.
3. Retrieve the Puppet public key and place it in a file on your node.
4. Use the RPM tool to import the public key, replacing <PUBLIC KEY FILE> with the path to the file containing the public key: `sudo rpm --import PUBKEY <PUBLIC KEY FILE>`

   The RPM tool doesn't output anything if the command is successful.
5. Use the RPM tool to check the signature of a downloaded RPM package: `sudo rpm -vK <RPM_FILE_NAME>`

   The embedded signature is verified and displays `OK`:

   ```
    puppet-agent-1.5.1-1.el6.x86_64.rpm:
        Header V4 RSA/SHA512 Signature, key ID ef8d349f: OK
        Header SHA1 digest: OK (95b492a1fff452d029aaeb59598f1c78dbfee0c5)
        V4 RSA/SHA512 Signature, key ID ef8d349f: OK
        MD5 digest: OK (4878909ccdd0af24fa9909790dd63a12)
   ```

### Verify a macOS `puppet-agent` package

`puppet-agent` packages for macOS are signed with a developer ID and certificate. You can verify the package signature using the `pkgutil` tool or the installer.

Use one of these methods to verify the package signature:

- Download and mount the `puppet-agent` disk image, and then use the `pkgutil` tool to check the package's signature:

  ```
  pkgutil --check-signature /Volumes/puppet-agent-<AGENT-
  VERSION>-1.osx10.10/puppet-agent-<AGENT-VERSION>-1-installer.pkg
  ```

  The tool confirms the signature and outputs fingerprints for each certificate in the chain:

  ```
  Package "puppet-agent-<AGENT-VERSION>-1-installer.pkg":
     Status: signed by a certificate trusted by macOS
  ```

```
    Certificate Chain:
     1. Developer ID Installer: PUPPET LABS, INC. (VKGLGN2B6Y)
        SHA1 fingerprint: AF 91 BF B7 7E CF 87 9F A8 0A 06 C3 03 5A B4 C7
11 34 0A 6F

-----------------------------------------------------------------------------
     2. Developer ID Certification Authority
        SHA1 fingerprint: 3B 16 6C 3B 7D C4 B7 51 C9 FE 2A FA B9 13 56 41
E3 88 E1 86

-----------------------------------------------------------------------------
     3. Apple Root CA
        SHA1 fingerprint: 61 1E 5B 66 2C 59 3A 08 FF 58 D1 4A E2 24 52 D1
98 DF 6C 60
```

- When you install the package, click the lock icon in the top right corner of the installer.

  The installer displays details about the package's certificate.

# Installing Puppet

For the agent-master architecture, you must install Puppet Server and, optionally, PuppetDB.

Enable the Puppet platform repository on page 48.

## Install Puppet Server

Puppet Server is a required application that runs on the Java Virtual Machine (JVM) on the master.

In addition to hosting endpoints for the certificate authority service, Puppet Server also powers the catalog compiler, which compiles configuration catalogs for agent nodes, using Puppet code and various other data sources.

Follow the steps to install Puppet Server.

## Install PuppetDB

You can optionally install PuppetDB to enable extra features, including enhanced queries and reports about your infrastructure.

Follow the steps to install PuppetDB .

# Installing agents

You can install agents on *nix, Windows, or macOS.

After installing agents, you must sign their certificates. For details, see Managing certificate signing requests on the command line on page 56

## Install *nix agents

You can install *nix agents using an install script.

Enable the Puppet platform repository on page 48.

1. Install the agent using the command appropriate to your environment.
    - Yum:

      ```
      sudo yum install puppet-agent
      ```
    - Apt:

      ```
      sudo apt-get install puppet-agent
      ```
    - Zypper:

      ```
      sudo zypper install puppet-agent
      ```
2. Start the Puppet service: `sudo /opt/puppetlabs/bin/puppet resource service puppet ensure=running enable=true`

## Install Windows agents

You can install Windows agents graphically or from the command line using an .msi package.

### Install Windows agents with the .msi package

Use the Windows .msi package if you need to specify agent configuration details during installation, or if you need to install Windows agents locally without internet access.

Download the .msi package.

### Install Windows agents with the installer

Use the MSI installer for a more automated installation process. The installer can configure `puppet.conf`, create CSR attributes, and configure the agent to talk to your master.

1. Run the installer as administrator.
2. When prompted, provide the hostname of your master, for example `puppet`.

### Install Windows agents using `msiexec` from the command line

Install the MSI manually from the the command line if you need to customize `puppet.conf`, CSR attributes, or certain agent properties.

On the command line of the node that you want to install the agent on, run the install command:

```
msiexec /qn /norestart /i <PACKAGE_NAME>.msi
```

**Tip:** You can specify `/l*v install.txt` to log the progress of the installation to a file.

### MSI properties

If you install Windows agents from the command line using the .msi package, you can optionally specify these properties.

**Important:** If you set a non-default value for `PUPPET_MASTER_SERVER`, `PUPPET_CA_SERVER`, `PUPPET_AGENT_CERTNAME`, or `PUPPET_AGENT_ENVIRONMENT`, the installer replaces the existing value in `puppet.conf` and re-uses the value at upgrade unless you specify a new value. Therefore, if you've customized these properties, don't change the setting directly in `puppet.conf`; instead, re-run the installer and set a new value at installation.

| Property | Definition | Setting in `pe.conf` | Default |
|---|---|---|---|
| `INSTALLDIR` | Location to install Puppet and its dependencies. | n/a | • 32-bit — `C:\Program Files\Puppet Labs\Puppet`<br>• 64-bit — `C:\Program Files\Puppet Labs\Puppet` |
| `PUPPET_MASTER_SERVER` | Hostname where the master can be reached. | `server` | `puppet` |
| `PUPPET_CA_SERVER` | Hostname where the CA master can be reached, if you're using multiple masters and only one of them is acting as the CA. | `ca_server` | Value of `PUPPET_MASTER_SERVER` |
| `PUPPET_AGENT_CERTNAME` | Node's certificate name, and the name it uses when requesting catalogs.<br><br>For best compatibility, limit the value of `certname` to lowercase letters, numbers, periods, underscores, and dashes. | `certname` | Value of `facter fdqn` |
| `PUPPET_AGENT_ENVIRONMENT` | Node's environment.<br><br>**Note:** If a value for the `environment` variable already exists in `puppet.conf`, specifying it during installation does not override that value. | `environment` | `production` |

| Property | Definition | Setting in `pe.conf` | Default |
|---|---|---|---|
| `PUPPET_AGENT_STARTUP_MODE` | Whether and how the agent service is allowed to run. Allowed values are:<br>• `Automatic` — Agent starts up when Windows starts and remains running in the background.<br>• `Manual` — Agent can be started in the services console or with `net start` on the command line.<br>• `Disabled` — Agent is installed but disabled. You must change its startup type in the services console before you can start the service. | n/a | `Automatic` |
| `PUPPET_AGENT_ACCOUNT_USER` | Windows user account the agent service uses. This property is useful if the agent needs to access files on UNC shares, because the default `LocalService` account can't access these network resources.<br><br>The user account must already exist, and can be a local or domain user. The installer allows domain users even if they have not accessed the machine before. The installer grants `Logon as Service` to the user, and if the user isn't already a local administrator, the installer adds it to the `Administrators` group.<br><br>This property must be combined with `PUPPET_AGENT_ACCOUNT_PASSWORD` and `PUPPET_AGENT_ACCOUNT_DOMAIN`. | n/a | `LocalSystem` |
| `PUPPET_AGENT_ACCOUNT_PASSWORD` | Password for the agent's user account. | n/a | No Value |
| `PUPPET_AGENT_ACCOUNT_DOMAIN` | Domain of the agent's user account. | n/a | `.` |

| Property | Definition | Setting in pe.conf | Default |
|---|---|---|---|
| REINSTALLMODE | A default MSI property used to control the behavior of file copies during installation.<br><br>**Important:** If you need to downgrade agents, use REINSTALLMODE=amus when calling msiexec.exe at the command line to prevent removing files that the application needs. | n/a | amus as of puppet-agent 1.10.10 and puppet-agent 5.3.4<br><br>omus in prior releases |

To install the agent with the master at `puppet.acme.com`:

```
msiexec /qn /norestart /i puppet.msi PUPPET_MASTER_SERVER=puppet.acme.com
```

To install the agent to a domain user `ExampleCorp\bob`:

```
msiexec /qn /norestart /i puppet-<VERSION>.msi
 PUPPET_AGENT_ACCOUNT_DOMAIN=ExampleCorp PUPPET_AGENT_ACCOUNT_USER=bob
 PUPPET_AGENT_ACCOUNT_PASSWORD=password
```

### Upgrading or downgrading between 32-bit and 64-bit Puppet on Windows

If necessary, you can upgrade or downgrade between 32-bit and 64-bit Puppet on Windows nodes.

#### Upgrading to 64-bit

To upgrade from 32-bit to 64-bit Puppet, simply install 64-bit Puppet. You don't need to uninstall the 32-bit version first.

The installer specifically stores information in different areas of the registry to allow rolling back to the 32-bit agent.

#### Downgrading to 32-bit

If you need to replace a 64-bit version of Puppet with a 32-bit version, you must uninstall Puppet before installing the new package.

You can uninstall Puppet through the **Add or Remove Programs** interface or from the command line.

To uninstall Puppet from the command line, you must have the original MSI file or know the ProductCode of the installed MSI:

```
msiexec /qn /norestart /x puppet-agent-1.3.0-x64.msi
msiexec /qn /norestart /x <PRODUCT CODE>
```

When you uninstall Puppet, the uninstaller removes the Puppet program directory, agent services, and all related registry keys. It leaves the $confdir, $codedir, and $vardir intact, including any SSL keys. To completely remove Puppet from the system, manually delete these directories.

## Install macOS agents

You can install macOS agents from Finder or from the command line.

Download the appropriate agent tarball.

**Important:** For macOS agents, the certname is derived from the name of the machine (such as My-Example-Mac). To prevent installation issues, make sure the name of the node uses lowercases letters. If you don't want to change your computer's name, you can enter the agent certname in all lowercase letters when prompted by the installer.

### Install macOS agents from Finder

You can use Finder to install the agent on your macOS machine.

1. Open the agent package .dmg and click the installer .pkg.
2. Follow prompts in the installer dialog.

   You must include the master hostname and the agent certname.

### Install macOS agents from the command line

You can use the command line to install the agent on a macOS machine.

1. SSH into the node as a root or sudo user.
2. Mount the disk image: `sudo hdiutil mount <DMGFILE>`

   A line appears ending with `/Volumes/puppet-agent-VERSION`. This directory location is the mount point for the virtual volume created from the disk image.
3. Change to the directory indicated as the mount point in the previous step, for example: `cd /Volumes/puppet-agent-VERSION`
4. Install the agent package: `sudo installer -pkg puppet-agent-installer.pkg -target /`
5. Verify the installation: `/opt/puppetlabs/bin/puppet --version`
6. Configure the agent to connect to the master: `/opt/puppetlabs/bin/puppet config set server <MASTER_HOSTNAME>`
7. Configure the agent certname: `/opt/puppetlabs/bin/puppet config set certname <AGENT_CERTNAME>`

## Adding executables to your PATH

To run interactive Puppet commands, you must either add their location to your `PATH` or execute them using their full path.

The location for Puppet executables is `/opt/puppetlabs/bin/`, which is not in your `PATH` environment variable by default.

To add the executable location to your `PATH` for your current terminal session, run:

```
export PATH=/opt/puppetlabs/bin:$PATH
```

Alternatively, you can add this location wherever you configure your `PATH`, such as your `.profile` or `.bashrc` configuration files.

## Managing certificate signing requests on the command line

You can view, approve, and reject node requests using the command line.

To view pending node requests on the command line:

```
$ sudo puppetserver ca list
```

To sign a pending request:

```
$ sudo puppetserver ca sign <NAME>
```

To sign pending requests for nodes with DNS altnames:

```
$ sudo puppetserver ca sign (<HOSTNAME> or --all) --allow-dns-alt-names
```

**Related information**

Puppet Server CA commands on page 486
Puppet Server has a `puppetserver ca` command that performs certificate authority (CA) tasks like signing and revoking certificates. Most of its actions are performed by making HTTP requests to Puppet Server's CA API, specifically the `certificate_status` endpoint. You must have Puppet Server running in order to sign or revoke certs.

### Configuring agents

You can add additional configuration to agents by editing `/etc/puppetlabs/puppet/puppet.conf` directly, or by using the `puppet config set` sub-command, which edits `puppet.conf` automatically.

For example, to point the agent at a master called `master.example.com`, run `puppet config set server master.example.com`. This command adds the setting `server = puppetmaster.example.com` to the `[main]` section of `puppet.conf`.

To set the certname for the agent, run `/opt/puppetlabs/bin/puppet config set certname agent.example.com`.

## Upgrading

To upgrade your deployment, you must upgrade both the infrastructure components and agents.

The order in which you upgrade components is important. Always upgrade Puppet Server and PuppetDB simultaneously, including the `puppetdb-termini` package on Puppet Server nodes, and always upgrade them before you upgrade agent nodes. Do not run different major versions on your Puppet masters (including Server) and PuppetDB nodes.

**Important:** These instructions cover upgrades in the versions 5 and 6 series. For instructions on upgrading from version 3.8.x, see previous versions of the documentation.

### Upgrade Puppet Server

Upgrade Puppet Server to adopt features and functionality of newer versions.

Upgrading the `puppetserver` package effectively upgrades Puppet Server. The `puppetserver` package, in turn, depends on the `puppet-agent` package, and your node's package manager automatically upgrades `puppet-agent` if the new version of `puppetserver` requires it.

**Important:** During an upgrade, Puppet Server doesn't perform its usual functions, including maintaining your site's infrastructure. If you use a single master, plan the timing of your upgrade accordingly and avoid reconfiguring any managed servers until your master is back up. If you use multiple load-balanced servers, upgrade them individually to avoid downtime or problems synchronizing configurations.

1. On your Puppet Server node, run the command appropriate to your package installer:

   Yum:

   ```
   yum update puppetserver
   ```

   Apt:

   ```
   apt-get update
   apt-get install --only-upgrade puppetserver
   ```

2. If you pinned Puppet packages to a specific version, remove the pins.

   For yum packages locked with the `versionlock` plugin, edit `/etc/yum/pluginconf.d/versionlock.list` to remove the lock.

   On apt systems, remove `.pref` files from `/etc/apt/preferences.d/` that pin packages, and use the `apt-mark unhold` command on each held package.

## Upgrade agents

Regularly upgrade agents to keep your systems running smoothly.

Upgrade Puppet Server.

### Upgrade *nix agents

To upgrade *nix nodes, use the `update` command.

On the agent node, run the command appropriate to your package installer:

Yum:

```
yum update puppet-agent
```

Apt:

```
apt-get update
apt-get install --only-upgrade puppet-agent
```

### Upgrade Windows agents

To upgrade Windows agents, reinstall the agent using the installation instructions. You don't need to uninstall the agent before reinstalling unless you’re upgrading from 32-bit Puppet to the 64-bit version.

### Upgrade macOS agents

Use the `puppet resource` command to upgrade macOS agents.

Download the appropriate agent tarball.

Use the package resource provider for macOS to install the agent from a disk image:

```
sudo puppet resource package "<NAME>.dmg" ensure=present source=<FULL PATH
 TO DMG>
```

## Upgrade PuppetDB

PuppetDB can be upgraded independently of your master and agent nodes.

1. Follow these steps, depending on whether you want to automate upgrade or manually upgrade.
   - To automate upgrade, specify the `version` parameter of the `puppetlabs/puppetdb` module’s `puppetdb::globals` class.
   - To manually upgrade, on the PuppetDB node, run the command appropriate to your package installer:

     Yum:

     ```
     yum update puppetdb
     ```

     Apt:

     ```
     apt-get update
     apt-get install --only-upgrade puppetdb
     ```

2. On your master, upgrade the `puppetdb-termini` package by running the command appropriate to your package installer:

   Yum:

   ```
   yum update puppetdb-termini
   ```

   Apt:

   ```
   apt-get update
   apt-get install --only-upgrade puppetdb-termini
   ```

# Configuring Puppet

You can configure Puppet's commands and services extensively, and its settings are specified in a variety of places.

- Puppet settings on page 59

Customize Puppet settings in the main configuration file, called `puppet.conf`.

- Key configuration settings on page 62

Puppet has about 200 settings, all of which are listed in the configuration reference. Most of the time, you interact with only a couple dozen of them. This page lists the most important ones, assuming that you're okay with default values for things like the port Puppet uses for network traffic. See the configuration reference for more details on each.

- Puppet's configuration files on page 65

Puppet settings can be configured in the main config file, called `puppet.conf`. There are several additional configuration files, for new settings and ones that need to be in separate files with complex data structures.

- Adding file server mount points on page 76

Puppet Server includes a file server for transferring static file content to agents. If you need to serve large files that you don't want to store in source control or distribute with a module, you can make a custom file server mount point and let Puppet serve those files from another directory.

- Checking the values of settings on page 79

Puppet settings are highly dynamic, and their values can come from several different places. To see the actual settings values that a Puppet service uses, run the `puppet config print` command.

- Editing settings on the command line on page 81

Puppet loads most of its settings from the `puppet.conf` config file. You can edit this file directly, or you can change individual settings with the `puppet config set` command.

- Complete list of settings (configuration reference)
- Settings that differ under Puppet Server

## Puppet settings

Customize Puppet settings in the main configuration file, called `puppet.conf`.

When Puppet documentation mentions “settings,” it usually means the main settings. These are the settings that are listed in the configuration reference. They are valid in `puppet.conf` and available for use on the command line. These settings configure nearly all of Puppet’s core features.

However, there are also several additional configuration files — such as `auth.conf` and `puppetdb.conf`. These files exist for several reasons:

- The main settings support only a few types of values. Some things just can’t be configured without complex data structures, so they needed separate files. (Authorization rules and custom CSR attributes are in this category.)
- Puppet doesn’t allow extensions to add new settings to `puppet.conf`. This means some settings that are supposed to be main settings (such as the PuppetDB server) can’t be.

### Puppet Server configuration

Puppet Server honors almost all settings in `puppet.conf` and picks them up automatically. However, for some tasks, such as configuring the webserver or an external Certificate Authority, there are Puppet Server-specific configuration files and settings.

For more information, see Puppet Server: Configuration.

### Settings are loaded on startup

When a Puppet command or service starts up, it gets values for all of its settings. Any of these settings can change the way that command or service behaves.

A command or service reads its settings only one time. If you need to reconfigured it, you must restart the service or run the command again after changing the setting.

### Settings on the command line

Settings specified on the command line have top priority and always override settings from the config file. When a command or service is started, you can specify any setting as a command line option.

Settings require two hyphens and the name of the setting on the command line:

```
$ sudo puppet agent --test --noop --certname temporary-name.example.com
```

### Basic settings

For most settings, you specify the option and follow it with a value. An equals sign between the two (`=`) is optional, and you can optionally put values in quotes.

All three of these are equivalent to setting `certname = temporary-name.example.com` in `puppet.conf`.

```
--certname=temporary-name.example.com
```

```
--certname temporary-name.example.com
```

```
--certname "temporary-name.example.com"
```

### Boolean settings

Settings whose only valid values are `true` and `false`, use a shorter format. Specifying the option alone sets the setting to `true`. Prefixing the option with `no-` sets it to false.

This means:

- `--noop` is equivalent to setting `noop = true` in `puppet.conf`.
- `--no-noop` is equivalent to setting `noop = false` in `puppet.conf`.

### Default values

If a setting isn't specified on the command line or in `puppet.conf`, it falls back to a default value. Default values for all settings are listed in the configuration reference.

Some default values are based on other settings — when this is the case, the default is shown using the other setting as a variable (similar to `$ssldir/certs`).

## Configuring locale settings

Puppet supports locale-specific strings in output, and it detects your locale from your system configuration. This provides localized strings, report messages, and log messages for the locale’s language when available.

Upon startup, Puppet looks for a set of environment variables on *nix systems, or the code page setting on Windows. When Puppet finds one that is set, it uses that locale whether it is run from the command line or as a service.

For help setting your operating system locale or adding new locales, consult its documentation. This section covers setting the locale for Puppet services.

### Checking your locale settings on *nix and macOS

To check your current locale settings, run the `locale` command. This outputs the settings used by your current shell.

```
$ locale
LANG="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_CTYPE="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_ALL=
```

To see which locales are supported by your system, run `locale -a`, which outputs a list of available locales. Note that Puppet might not have localized strings for every available locale.

To check the current status of environment variables that might conflict with or override your locale settings, use the `set` command. For example, this command lists the set environment variables and searches for those containing `LANG` or `LC_`:

```
sudo set | egrep 'LANG|LC_'
```

### Checking your locale settings on Windows

To check your current locale setting, run the `Get-WinSystemLocale` command from PowerShell.

```
PS C:\> Get-WinSystemLocale
LCID             Name             DisplayName
----             ----             -----------
1033             en-US            English (United States)
```

To check your system’s current code page setting, run the `chcp` command.

### Setting your locale on *nix with an environment variable

You can use environment variables to set your locale for processes started on the command line. For most Linux distributions, set the `LANG` variable to your preferred locale, and the `LANGUAGE` variable to an empty string. On SLES, also set the `LC_ALL` variable to an empty string.

For example, to set the locale to Japanese for a terminal session on SLES:

```
export LANG=ja_JP.UTF-8
export LANGUAGE=''
export LC_ALL=''
```

To set the locale for the Puppet agent service, you can add these `export` statements to:

- `/etc/sysconfig/puppet` on RHEL and its derivatives

- `/etc/default/puppet` on Debian, Ubuntu, and their derivatives

  After updating the file, restart the Puppet service to apply the change.

### Setting your locale for the Puppet agent service on macOS

To set the locale for the Puppet agent service on macOS, update the `LANG` setting in the `/Library/LaunchDaemons/com.puppetlabs.puppet.plist` file.

```
<dict>
        <key>LANG</key>
        <string>ja_JP.UTF-8</string>
</dict>
```

After updating the file, restart the Puppet service to apply the change.

### Setting your locale on Windows

On Windows, Puppet uses the `LANG` environment variable if it is set. If not, it uses the configured region, as set in the Administrator tab of the Region control panel.

On Windows 10, you can use PowerShell to set the system locale:

```
Set-WinSystemLocale en-US
```

### Disabling internationalized strings

Use the optional Boolean `disable_i18n` setting to disable the use of internationalized strings. You can configure this setting in `puppet.conf`. If set to `true`, Puppet disables localized strings in log messages, reports, and parts of the command line interface. This can improve performance when using Puppet modules, especially if environment caching is disabled, and even if you don't need localized strings or the modules aren't localized. This setting is `false` by default in open source Puppet.

If you're experiencing performance issues, configure this setting in the `[master]` section of the Puppet master's `puppet.conf` file. To force unlocalized messages, which are in English by default, configure this section in a node's `[main]` or `[user]` sections of `puppet.conf`.

## Key configuration settings

Puppet has about 200 settings, all of which are listed in the configuration reference. Most of the time, you interact with only a couple dozen of them. This page lists the most important ones, assuming that you're okay with default values for things like the port Puppet uses for network traffic. See the configuration reference for more details on each.

There are a lot of settings that are rarely useful but still make sense, but there are also at least a hundred that are not configurable at all. This is a Puppet design choice. Because of the way Puppet code is arranged, the settings system is the easiest way to publish global constants that are dynamically initialized on startup. This means a lot of things have been introduced to Puppet as configurable settings regardless of whether they needed to be configurable.

For a full list of Puppet settings, see the configuration reference.

## Settings for agents (all nodes)

The following settings for agents are listed approximately in order of importance. Most of these can go in either `[main]` or `[agent]` sections, or be specified on the command line.

### Basics

- `server` — The master server to request configurations from. Defaults to `puppet`. Change it if that's not your server's name.
    - `ca_server` and `report_server` — If you're using multiple masters, you'll need to centralize the CA. One of the ways to do this is by configuring `ca_server` on all agents. See Scaling Puppet Server with compile masters for more details. The `report_server` setting works the same way, although whether you need to use it depends on how you're processing reports.
- `certname` — The node's certificate name, and the unique identifier it uses when requesting catalogs. Defaults to the fully qualified domain name.
    - For best compatibility, you should limit the value of `certname` to only use letters, numbers, periods, underscores, and dashes. That is, it should match `/\A[a-z0-9._-]+\Z/`.
    - The special value `ca` is reserved, and can't be used as the certname for a normal node.
- `environment` — The environment to request when contacting the master. It's only a request, though; the master's ENC can override this if it chooses. Defaults to `production`.
- `sourceaddress` — The address on a multihomed host to use for the agent's communication with the master server.

**Note:** Although it's possible to set something other than the certname as the node name (using either the `node_name_fact` or `node_name_value` setting), we don't generally recommend it. It allows you to re-use one node certificate for many nodes, but it reduces security, makes it harder to reliably identify nodes, and can interfere with other features. Setting a non-certname node name is not officially supported in Puppet Enterprise.

### Run behavior

These settings affect the way Puppet applies catalogs:

- `noop` — If enabled, the agent won't make any changes to the node. Instead, it looks for changes that would be made if the catalog were applied, and report to the master about what it would have done. This can be overridden per-resource with the `noop` metaparameter.
- `priority` — Allows you to make the agent share CPU resources so that other applications have access to processing power while agent is applying a catalog.
- `report` — Indicates whether to send reports. Defaults to true, and under most circumstances should be left there.
- `tags` — Lets you limit the Puppet run to include only those resources with certain tags.
- `trace`, `profile`, `graph`, and `show_diff` — Tools for debugging or learning more about an agent run. Useful when combined with the `--test` and `--debug` command options.
- `usecacheonfailure` — Indicates whether to fall back to the last known good catalog if the master fails to return a good catalog. The default behavior is usually what you want, but you might have a reason to disable it.
- `ignoreschedules` — If you use schedules, this can be useful when doing an initial Puppet run to set up new nodes.
- `prerun_command` and `postrun_command` — Commands to run on either side of a Puppet run.

### Service behavior

These settings affect the way Puppet agent acts when running as a long-lived service:

- `runinterval` — How often to do a Puppet run, when running as a service.
- `waitforcert` — Whether to keep trying if the agent can't initially get a certificate. The default behavior is good, but you might have a reason to disable it.

### Useful when running agent from cron

- `splay` and `splaylimit` — Together, these allow you to spread out agent runs. When running the agent as a daemon, the services usually have been started far enough out of sync to make this a non-issue, but it's useful with cron agents. For example, if your agent cron job happens on the hour, you could set `splay = true` and `splaylimit = 60m` to keep the master from getting briefly overworked and then left idle for the next 50 minutes.
- `daemonize` — Whether to daemonize. Set this to false when running the agent from cron.
- `onetime` — Whether to exit after finishing the current Puppet run. Set this to true when running the agent from cron.

For more information on these settings, see the configuration reference.

## Settings for master servers

Many of these settings are also important for standalone Puppet apply nodes, because they act as their own master. These settings should usually go in `[master]`. However, if you're using Puppet apply in production, put them in `[main]` instead.

### Basics

- `dns_alt_names` — A list of hostnames the server is allowed to use when acting as a master. The hostname your agents use in their `server` setting must be included in either this setting or the master's `certname` setting. Note that this setting is only used when initially generating the master's certificate — if you need to change the DNS names, you must:

    1. Turn off the Puppet Server service (or your Rack server).
    2. Run: `sudo puppetserver ca clean <MASTER'S CERTNAME>`
    3. Run: `sudo puppetserver ca generate <MASTER'S CERTNAME> --dns-alt-names <ALT NAME 1>,<ALT NAME 2>,...`
    4. Re-start the Puppet Server service.
- `environment_timeout` — For better performance, you can set this to `unlimited` and make refreshing the master a part of your standard code deployment process.
- `environmentpath` — Controls where Puppet finds directory environments. For more information on environments, see Creating environments.
- `basemodulepath` — A list of directories containing Puppet modules that can be used in all environments. See modulepath for details.
- `reports` — Which report handlers to use. For a list of available report handlers, see the report reference. You can also write your own report handlers. Note that the report handlers might require settings of their own.

### Puppet Server related settings

Puppet Server has its own configuration files; consequently, there are several settings in `puppet.conf` that Puppet Server ignores.

- `puppet-admin` — Settings to control which authorized clients can use the admin interface.
- `jruby-puppet` — Provides details on tuning JRuby for better performance.
- `JAVA_ARGS` — Instructions on tuning the Puppet Server memory allocation.

### Extensions

These features configure add-ons and optional features:

- `node_terminus` and `external_nodes` — The ENC settings. If you're using an ENC, set these to `exec` and the path to your ENC script, respectively.
- `storeconfigs` and `storeconfigs_backend` — Used for setting up PuppetDB. See the PuppetDB docs for details.

- `catalog_terminus` — This can enable the optional static compiler. If you have lots of `file` resources in your manifests, the static compiler lets you sacrifice some extra CPU work on your master to gain faster configuration and reduced HTTPS traffic on your agents. See the indirection reference for details.

### CA settings

- `ca_ttl` — How long newly signed certificates should be valid for. Deprecated.
- `autosign` — Whether and how to autosign certificates. See Autosigning for detailed information.

For more information on these settings, see the configuration reference.

## Puppet's configuration files

Puppet settings can be configured in the main config file, called `puppet.conf`. There are several additional configuration files, for new settings and ones that need to be in separate files with complex data structures.

- puppet.conf: The main config file on page 65

The `puppet.conf` file is Puppet's main config file. It configures all of the Puppet commands and services, including Puppet agent, Puppet master, Puppet apply, and `puppetserver ca`. Nearly all of the settings listed in the configuration reference can be set in `puppet.conf`.

- environment.conf: Per-environment settings on page 68

Any environment can contain an `environment.conf` file. This file can override several settings whenever the master is serving nodes assigned to that environment.

- fileserver.conf: Custom fileserver mount points on page 70

The `fileserver.conf` file configures custom static mount points for Puppet's file server. If custom mount points are present, `file` resources can access them with their `source` attributes.

- puppetdb.conf: PuppetDB server locations on page 71

The `puppetdb.conf` file configures how Puppet connects to one or more servers. It is only used if you are using PuppetDB and have connected your master to it.

- hiera.yaml: Data lookup configuration

- autosign.conf: Basic certificate autosigning on page 71

The `autosign.conf` file can allow certain certificate requests to be automatically signed. It is only valid on the CA Puppet master server; a Puppet master not serving as a CA does not use `autosign.conf`.

- csr_attributes.yaml: Certificate extensions on page 72

The `csr_attributes.yaml` file defines custom data for new certificate signing requests (CSRs).

- custom_trusted_oid_mapping.yaml: Short names for cert extension OIDs on page 74

The `custom_trusted_oid_mapping.yaml` file lets you set your own short names for certificate extension object identifiers (OIDs), which can make the `$trusted` variable more useful.

- device.conf: Network hardware access on page 75

The `puppet-device` subcommand retrieves catalogs from the Puppet master and applies them to remote devices. Devices to be managed by the `puppet-device` subcommand are configured in `device.conf`.

- routes.yaml: Advanced plugin routing on page 76

The `routes.yaml` file overrides configuration settings involving indirector termini, and allows termini to be set in greater detail than `puppet.conf` allows.

### `puppet.conf`: The main config file

The `puppet.conf` file is Puppet's main config file. It configures all of the Puppet commands and services, including Puppet agent, Puppet master, Puppet apply, and `puppetserver ca`. Nearly all of the settings listed in the configuration reference can be set in `puppet.conf`.

It resembles a standard INI file, with a few syntax extensions. Settings can go into application-specific sections, or into a `[main]` section that affects all applications.

For a complete list of Puppet's settings, see the configuration reference.

### Location

The `puppet.conf` file is always located at `$confdir/puppet.conf`.

Although its location is configurable with the `config` setting, it can be set only on the command line. For example:

```
puppet agent -t --config ./temporary_config.conf
```

The location of the `confdir` depends on your operating system. See the confdir documentation for details.

### Examples

Example agent config:

```
[main]
certname = agent01.example.com
server = puppet
environment = production
runinterval = 1h
```

Example master config:

```
[main]
certname = puppetmaster01.example.com
server = puppet
environment = production
runinterval = 1h
strict_variables = true

[master]
dns_alt_names =
 puppetmaster01,puppetmaster01.example.com,puppet,puppet.example.com
reports = puppetdb
storeconfigs_backend = puppetdb
storeconfigs = true
environment_timeout = unlimited
```

### Format

The `puppet.conf` file consists of one or more config sections, each of which can contain any number of settings.

The file can also include comment lines at any point.

### Config sections

```
[main]
    certname = puppetmaster01.example.com
```

A config section is a group of settings. It consists of:

- Its name, enclosed in square brackets. The `[name]` of the config section must be on its own line, with no leading space.
- Any number of setting lines, which can be indented for readability.
- Any number of empty lines or comment lines

As soon as a new config section `[name]` appears in the file, the former config section is closed and the new one begins. A given config section only occurs one time in the file.

Puppet uses four config sections:

- `main` is the global section used by all commands and services. It can be overridden by the other sections.

- `master` is used by the Puppet master service and the Puppet Server `ca` command.
- `agent` is used by the Puppet agent service.
- `user` is used by the Puppet apply command, as well as many of the less common Puppet subcommands.

Puppet prefers to use settings from one of the three application-specific sections (`master`, `agent`, or `user`). If it doesn't find a setting in the application section, it uses the value from `main`. (If `main` doesn't set one, it falls back to the default value.)

**Note:** Puppet Server ignores some config settings. It honors almost all settings in `puppet.conf` and picks them up automatically. However, some Puppet Server settings differ from a Ruby Puppet master's `puppet.conf` settings.

### Comment lines

```
# This is a comment.
```

Comment lines start with a hash sign (#). They can be indented with any amount of leading space.

Partial-line comments such as `report = true # this enables reporting` are not allowed, and the intended comment is treated as part of the value of the setting. To be treated as a comment, the hash sign must be the first non-space character on the line.

### Setting lines

```
certname = puppetmaster01.example.com
```

A setting line consists of:

- Any amount of leading space (optional).
- The name of a setting.
- An equals sign (`=`), which can optionally be surrounded by any number of spaces.
- A value for the setting.

### Special types of values for settings

Generally, the value of a setting is a single word. However, listed below are a few special types of values.

List of words: Some settings (like reports) can accept multiple values, which are specified as a comma-separated list (with optional spaces after commas). Example: `report = http,puppetdb`

Paths: Some settings (like `environmentpath`) take a list of directories. The directories are separated by the system path separator character, which is colon (`:`) on *nix platforms and semicolon (`;`) on Windows.

```
# *nix version:
environmentpath = $codedir/special_environments:$codedir/environments
# Windows version:
environmentpath = $codedir/environments;C:\ProgramData\PuppetLabs\code
\environment
```

Path lists are ordered;Puppet always checks the first directory first, then moves on to the others if it doesn't find what it needs.

Files or directories: Settings that take a single file or directory (like `ssldir`) can accept an optional hash of permissions. When starting up, Puppet enforces those permissions on the file or directory.

We do not recommend you do this because the defaults are good for most users. However, if you need to, you can specify permissions by putting a hash like this after the path:

```
ssldir = $vardir/ssl {owner = service, mode = 0771}
```

The allowed keys in the hash are`owner`, `group`, and `mode`. There are only two valid values for the `owner` and `group` keys:

- `root` — the root or Administrator user or group owns the file.
- `service` — the user or group that the Puppet service is running as owns the file. The service's user and group are specified by the `user` and `group` settings. On a master running open source Puppet, these default to `puppet`; on Puppet Enterprise they default to `pe-puppet`.

### Interpolating variables in settings

The values of settings are available as variables within `puppet.conf`, and you can insert them into the values of other settings. To reference a setting as a variable, prefix its name with a dollar sign (`$`):

```
ssldir = $vardir/ssl
```

Not all settings are equally useful; there's no real point in interpolating`$ssldir` into `basemodulepath`, for example. We recommend that you use only the following variables:

- `$codedir`
- `$confdir`
- `$vardir`

## `environment.conf`: Per-environment settings

Any environment can contain an `environment.conf` file. This file can override several settings whenever the master is serving nodes assigned to that environment.

### Location

Each `environment.conf` file is stored in an environment. It will be at the top level of its home environment, next to the `manifests` and `modules` directories.

For example, if your environments are in the default directory (`$codedir/environments`), the `test` environment's config file is located at `$codedir/environments/test/environment.conf`.

### Example

```
# /etc/puppetlabs/code/environments/test/environment.conf

# Puppet Enterprise requires $basemodulepath; see note below under
 "modulepath".
modulepath = site:dist:modules:$basemodulepath

# Use our custom script to get a git commit for the current state of the
 code:
config_version = get_environment_commit.sh
```

### Format

The `environment.conf` file uses the same INI-like format as `puppet.conf`, with one exception: it cannot contain config sections like `[main]`. All settings in `environment.conf` must be outside any config section.

### Relative paths in values

Most of the allowed settings accept file paths or lists of paths as their values.

If any of these paths are relative paths — that is, they start without a leading slash or drive letter — they are resolved relative to that environment's main directory.

For example, if you set `config_version = get_environment_commit.sh` in the `test` environment, Puppet uses the file at `/etc/puppetlabs/code/environments/test/get_environment_commit.sh`.

### Interpolation in values

The settings in `environment.conf` can use the values of other settings as variables (such as `$codedir`). Additionally, the `config_version` setting can use the special `$environment` variable, which gets replaced with the name of the active environment.

The most useful variables to interpolate into `environment.conf` settings are:

- `$basemodulepath` — useful for including the default module directories in the `modulepath` setting. We recommend Puppet Enterprise (PE) users include this in the value of `modulepath`, because PE uses modules in the `basemodulepath` to configure orchestration and other features.
- `$environment` — useful as a command line argument to your `config_version` script. You can interpolate this variable only in the `config_version` setting.
- `$codedir` — useful for locating files.

### Allowed settings

The `environment.conf` file can override these settings:

**`modulepath`**

The list of directories Puppet loads modules from.

If this setting isn't set, the `modulepath` for the environment is:

```
<MODULES DIRECTORY FROM
 ENVIRONMENT>:$basemodulepath
```

That is, Puppet adds the environment's modules directory to the value of the basemodulepath setting from `puppet.conf`, with the environment's modules getting priority. If the modules directory is empty of absent, Puppet only uses modules from directories in the basemodulepath. A directory environment never uses the global `modulepath` from `puppet.conf`.

**`manifest`**

The main manifest the Puppet master uses when compiling catalogs for this environment. This can be one file or a directory of manifests to be evaluated in alphabetical order. Puppet manages this path as a directory if one exists or if the path ends with a slash (`/`) or dot (`.`).

If this setting isn't set, Puppet uses the environment's `manifests` directory as the main manifest, even if it is empty or absent. A directory environment never uses the global `manifest` from `puppet.conf`.

**`config_version`**

A script Puppet can run to determine the configuration version.

Puppet automatically adds a config version to every catalog it compiles, as well as to messages in reports. The version is an arbitrary piece of data that can be used to identify catalogs and events.

| | |
|---|---|
| | You can specify an executable script that determines an environment's config version by setting `config_version` in its environment.conf file. Puppet runs this script when compiling a catalog for a node in the environment, and use its output as the config version.<br><br>**Note:** If you're using a system binary like `git rev-parse`, make sure to specify the absolute path to it. If `config_version` is set to a relative path, Puppet looks for the binary in the environment, not in the system's `PATH`.<br><br>If this setting isn't set, the config version is the time at which the catalog was compiled (as the number of seconds since January 1, 1970). A directory environment never uses the global `config_version` from `puppet.conf`. |
| **`environment_timeout`** | How long the Puppet master caches the data it loads from an environment. If present, this overrides the value of `environment_timeout` from puppet.conf. Unless you have a specific reason, we recommend only setting `environment_timeout` globally, in puppet.conf. We also don't recommend using any value other than `0` or `unlimited`.<br><br>For more information about configuring the environment timeout, see the timeout section of the Creating Environments page. |

## `fileserver.conf`: Custom fileserver mount points

The `fileserver.conf` file configures custom static mount points for Puppet's file server. If custom mount points are present, `file` resources can access them with their `source` attributes.

### When to use `fileserver.conf`

This file is necessary only if you are creating custom mount points.

Puppet automatically serves files from the `files` directory of every module, and most users find this sufficient. For more information, see Modules fundamentals. However, custom mount points are useful for things that you don't store in version control with your modules, like very large files and sensitive credentials.

### Location

The `fileserver.conf` file is located at `$confdir/fileserver.conf` by default. Its location is configurable with the `fileserverconfig` setting.

The location of the `confdir` depends on your operating system. See the confdir documentation for details.

### Example

```
# Files in the /path/to/files directory are served
# at puppet:///extra_files/.
[extra_files]
    path /etc/puppetlabs/puppet/extra_files
    allow *
```

This `fileserver.conf` file would create a new mount point named `extra_files`.

**CAUTION:** Always restrict write access to mounted directories. The file server follows any symlinks in a file server mount, including links to files that agent nodes shouldn't access (like SSL keys). When following symlinks, the file server can access any files readable by Puppet Server's user account.

### Format

`fileserver.conf` uses a one-off format that resembles an INI file without the equals `(=)` signs. It is a series of mount-point stanzas, where each stanza consists of:

- A `[mount_point_name]` surrounded by square brackets. This becomes the name used in `puppet:///` URLs for files in this mount point.
- A `path <PATH>` directive, where `<PATH>` is an absolute path on disk. This is where the mount point's files are stored.
- An `allow *` directive.

### Deprecated security directives

Before `auth.conf` existed, `fileserver.conf` could use `allow` and `deny` directives to control which nodes can access various files. This feature is now deprecated, and will be removed in a future release of Puppet.

Instead, you can use `auth.conf` to control access to mount points. See setting up mount points for more details and examples.

The only security directive present in `fileserver.conf` is an `allow *` directive for every mount point.

## `puppetdb.conf`: PuppetDB server locations

The `puppetdb.conf` file configures how Puppet connects to one or more servers. It is only used if you are using PuppetDB and have connected your master to it.

This configuration file is documented in the PuppetDB docs. See Configuring a Puppet/PuppetDB connection for details.

## `autosign.conf`: Basic certificate autosigning

The `autosign.conf` file can allow certain certificate requests to be automatically signed. It is only valid on the CA Puppet master server; a Puppet master not serving as a CA does not use `autosign.conf`.

**CAUTION:** Because any host can provide any certname when requesting a certificate, basic autosigning is insecure. Use it only when you fully trust any computer capable of connecting to the master.

Puppet also provides a policy-based autosigning interface using custom policy executables, which can be more flexible and secure than the `autosign.conf` whitelist but more complex to configure.

For more information, see the documentation about certificate autosigning.

### Location

Puppet looks for `autosign.conf` at `$confdir/autosign.conf` by default. To change this path, configure the autosign setting in the `[master]` section of `puppet.conf`.

The default confdir path depends on your operating system. See the confdir documentation for more information.

**Note:** The `autosign.conf` file must not be executable by the master's user account. If the `autosign` setting points to an executable file, Puppet instead treats it like a custom policy executable even if it contains a valid `autosign.conf` whitelist.

#### Format

The `autosign.conf` file is a line-separated list of certnames or domain name globs. Each line represents a node name or group of node names for which the CA Puppet master automatically signs certificate requests.

```
rebuilt.example.com
*.scratch.example.com
*.local
```

Domain name globs do not function as normal globs: an asterisk can only represent one or more subdomains at the front of a certname that resembles a fully qualified domain name (FQDN). If your certnames don't look like FQDNs, the `autosign.conf` whitelist might not be effective.

**Note:** The `autosign.conf` file can safely be an empty file or not-existent, even if the `autosign` setting is enabled. An empty or non-existent `autosign.conf` file is an empty whitelist, meaning that Puppet does not autosign any requests. If you create `autosign.conf` as a non-executable file and add certnames to it, Puppet then automatically uses the file to whitelist incoming requests without needing to modify `puppet.conf`.

To explicitly disable autosigning, set `autosign = false` in the `[master]` section of the CA Puppet master's `puppet.conf`, which disables CA autosigning even if `autosign.conf` or a custom policy executable exists.

### `csr_attributes.yaml`: Certificate extensions

The `csr_attributes.yaml` file defines custom data for new certificate signing requests (CSRs).

The `csr_attributes.yaml` file can set:

- CSR attributes (transient data used for pre-validating requests)
- Certificate extension requests (permanent data to be embedded in a signed certificate)

This file is only consulted when a new CSR is created, for example when an agent node is first attempting to join a Puppet deployment. It cannot modify existing certificates.

For information about using this file, see CSR attributes and certificate extensions.

#### Location

The `csr_attributes.yaml` file is located at `$confdir/csr_attributes.yaml` by default. Its location is configurable with the `csr_attributes` setting.

The location of the `confdir` depends on your operating system. See the confdir documentation for details.

#### Example

```
---
custom_attributes:
  1.2.840.113549.1.9.7: 342thbjkt82094y0uthhor289jnqthpc2290
extension_requests:
  pp_uuid: ED803750-E3C7-44F5-BB08-41A04433FE2E
  pp_image_name: my_ami_image
  pp_preshared_key: 342thbjkt82094y0uthhor289jnqthpc2290
```

#### Format

The `csr_attributes` file must be a YAML hash containing one or both of the following keys:

- `custom_attributes`
- `extension_requests`

The value of each key must also be a hash, where:

- Each key is a valid object identifier (OID). Note that Puppet-specific OIDs can optionally be referenced by short name instead of by numeric ID. In the example above, `pp_uuid` is a short name for a Puppet-specific OID.
- Each value is an object that can be cast to a string. That is, numbers are allowed but arrays are not.

### Allowed OIDs for custom attributes

Custom attributes can use any public or site-specific OID, with the exception of the OIDs used for core X.509 functionality. This means you can't re-use existing OIDs for things like subject alternative names.

One useful OID is the "challengePassword" attribute — `1.2.840.113549.1.9.7`. This is a rarely-used corner of X.509 which can be repurposed to hold a pre-shared key. The benefit of using this instead of an arbitrary OID is that it appears by name when using OpenSSL to dump the CSR to text; OIDs that `openssl req` can't recognize are displayed as numerical strings.

Also note that the Puppet-specific OIDs listed below can also be used in CSR attributes.

### Allowed OIDs for extension requests

Extension request OIDs **must** be under the "ppRegCertExt" (`1.3.6.1.4.1.34380.1.1`) or "ppPrivCertExt" (`1.3.6.1.4.1.34380.1.2`) OID arcs.

Puppet provides several registered OIDs (under "ppRegCertExt") for the most common kinds of extension information, as well as a private OID range ("ppPrivCertExt") for site-specific extension information. The benefits of using the registered OIDs are:

- They can be referenced in `csr_attributes.yaml` using their short names instead of their numeric IDs.
- When using Puppet tools to print certificate info, they appear using their descriptive names instead of their numeric IDs.

The private range is available for any information you want to embed into a certificate that isn't already in wide use elsewhere. It is completely unregulated, and its contents are expected to be different in every Puppet deployment.

The "ppRegCertExt" OID range contains the following OIDs.

| Numeric ID | Short name | Descriptive name |
|---|---|---|
| 1.3.6.1.4.1.34380.1.1.1 | `pp_uuid` | Puppet Node UUID |
| 1.3.6.1.4.1.34380.1.1.2 | `pp_instance_id` | Puppet Node Instance ID |
| 1.3.6.1.4.1.34380.1.1.3 | `pp_image_name` | Puppet Node Image Name |
| 1.3.6.1.4.1.34380.1.1.4 | `pp_preshared_key` | Puppet Node Preshared Key |
| 1.3.6.1.4.1.34380.1.1.5 | `pp_cost_center` | Puppet Node Cost Center Name |
| 1.3.6.1.4.1.34380.1.1.6 | `pp_product` | Puppet Node Product Name |
| 1.3.6.1.4.1.34380.1.1.7 | `pp_project` | Puppet Node Project Name |
| 1.3.6.1.4.1.34380.1.1.8 | `pp_application` | Puppet Node Application Name |
| 1.3.6.1.4.1.34380.1.1.9 | `pp_service` | Puppet Node Service Name |
| 1.3.6.1.4.1.34380.1.1.10 | `pp_employee` | Puppet Node Employee Name |
| 1.3.6.1.4.1.34380.1.1.11 | `pp_created_by` | Puppet Node `created_by` Tag |
| 1.3.6.1.4.1.34380.1.1.12 | `pp_environment` | Puppet Node Environment Name |
| 1.3.6.1.4.1.34380.1.1.13 | `pp_role` | Puppet Node Role Name |
| 1.3.6.1.4.1.34380.1.1.14 | `pp_software_version` | Puppet Node Software Version |
| 1.3.6.1.4.1.34380.1.1.15 | `pp_department` | Puppet Node Department Name |
| 1.3.6.1.4.1.34380.1.1.16 | `pp_cluster` | Puppet Node Cluster Name |

| Numeric ID | Short name | Descriptive name |
|---|---|---|
| 1.3.6.1.4.1.34380.1.1.17 | pp_provisioner | Puppet Node Provisioner Name |
| 1.3.6.1.4.1.34380.1.1.18 | pp_region | Puppet Node Region Name |
| 1.3.6.1.4.1.34380.1.1.19 | pp_datacenter | Puppet Node Datacenter Name |
| 1.3.6.1.4.1.34380.1.1.20 | pp_zone | Puppet Node Zone Name |
| 1.3.6.1.4.1.34380.1.1.21 | pp_network | Puppet Node Network Name |
| 1.3.6.1.4.1.34380.1.1.22 | pp_securitypolicy | Puppet Node Security Policy Name |
| 1.3.6.1.4.1.34380.1.1.23 | pp_cloudplatform | Puppet Node Cloud Platform Name |
| 1.3.6.1.4.1.34380.1.1.24 | pp_apptier | Puppet Node Application Tier |
| 1.3.6.1.4.1.34380.1.1.25 | pp_hostname | Puppet Node Hostname |

The “ppAuthCertExt” OID range contains the following OIDs:

| | | |
|---|---|---|
| 1.3.6.1.4.1.34380.1.3.1 | pp_authorization | Certificate Extension Authorization |
| 1.3.6.1.4.1.34380.1.3.13 | pp_auth_role | Puppet Node Role Name for Authorization |

## `custom_trusted_oid_mapping.yaml`: Short names for cert extension OIDs

The `custom_trusted_oid_mapping.yaml` file lets you set your own short names for certificate extension object identifiers (OIDs), which can make the `$trusted` variable more useful.

It is only valid on a Puppet master server. In Puppet apply, the compiler doesn’t add certificate extensions to `$trusted.`

### Certificate extensions

When a node requests a certificate, it can ask the CA to include some additional, permanent metadata in that cert. Puppet agent uses the `csr_attributes.yaml` file to decide what extensions to request.

If the CA signs a certificate with extensions included, those extensions are available as trusted facts in the top-scope `$trusted` variable. Your manifests or node classifier can then use those trusted facts to decide which nodes can receive which configurations.

By default, the Puppet-specific registered OIDs appear as keys with convenient short names in the `$trusted[extensions]` hash, and any other OIDs appear as raw numerical IDs. You can use the `custom_trusted_oid_mapping.yaml` file to map other OIDs to short names, which replaces the numerical OIDs in `$trusted[extensions]`.

Run `puppetserver ca print` to see changes made in `custom_trusted_oid_mapping.yaml` immediately without a restart.

For more information, see CSR attributes and certificate extensions, Trusted facts, The `csr_attributes.yaml` file.

### Limitations of OID mapping

Mapping OIDs in this file **only** affects the keys in the `$trusted[extensions]` hash. It does not affect what an agent can request in its `csr_attributes.yaml` file — anything but Puppet-specific registered extensions must still be numerical OIDs.

After setting custom OID mapping values and restarting puppetserver, you can reference variables using only the short name.

#### Location

The OID mapping file is located at `$confdir/custom_trusted_oid_mapping.yaml` by default. Its location is configurable with the `trusted_oid_mapping_file` setting.

The location of the `confdir` depends on your OS. See the confdir documentation for details.

#### Example

```
---
oid_mapping:
  1.3.6.1.4.1.34380.1.2.1.1:
    shortname: 'myshortname'
    longname: 'My Long Name'
  1.3.6.1.4.1.34380.1.2.1.2:
    shortname: 'myothershortname'
    longname: 'My Other Long Name'
```

#### Format

The `custom_trusted_oid_mapping.yaml` must be a YAML hash containing a single key called `oid_mapping`.

The value of the `oid_mapping` key must be a hash whose keys are numerical OIDs. The value for each OID must be a hash with two keys:

- `shortname` for the case-sensitive one-word name that is used in the `$trusted[extensions]` hash.
- `longname` for a more descriptive name (not used elsewhere).

### `device.conf`: Network hardware access

The `puppet-device` subcommand retrieves catalogs from the Puppet master and applies them to remote devices. Devices to be managed by the `puppet-device` subcommand are configured in `device.conf`.

For more information on Puppet device, see the Puppet device documentation.

#### Location

The `device.conf` file is located at `$confdir/device.conf` by default, and its location is configurable with the deviceconfig setting.

The location of `confdir` depends on your operating system. See the confdir documentation for details.

#### Format

The `device.conf` file is an INI-like file, with one section per device:

```
[device001.example.com]
type cisco
url ssh://admin:password@device001.example.com
debug
```

The section name specifies the `certname` of the device.

The values for the `type` and `url` properties are specific to each type of device.

The the optional `debug` property specifies transport-level debugging, and is limited to telnet and ssh transports.

For Cisco devices, the `url` is in the following format:

```
scheme://user:password@hostname/query
```

With:

- Scheme: either `ssh` or `telnet`
- user: optional connection username, depending on the device configuration
- password: connection password
- query: optional `?enable=` parameter whose value is the enable password

**Note:** Reserved non-alphanumeric characters in the `url` must be percent-encoded.

## `routes.yaml`: Advanced plugin routing

The `routes.yaml` file overrides configuration settings involving indirector termini, and allows termini to be set in greater detail than `puppet.conf` allows.

The `routes.yaml` file makes it possible to use certain extensions to Puppet, most notably PuppetDB. Usually you edit this file only to make changes that are explicitly specified by the setup instructions for an extension you are trying to install.

### Location

The `routes.yaml` file is located at `$confdir/routes.yaml` by default. Its location is configurable with the `route_file` setting.

The location of the `confdir` depends on your operating system. See the confdir documentation for details.

### Example

```
---
master:
  facts:
    terminus: puppetdb
    cache: yaml
```

### Format

The `routes.yaml` file is a YAML hash.

Each top level key is the name of a run mode (`master`, `agent`, or `user`), and its value is another hash.

Each key of the second-level hash is the name of an indirection, and its value is another hash.

The only keys allowed in the third-level hash are `terminus` and `cache`. The value of each of these keys is the name of a valid terminus for the indirection named above.

# Configuring Puppet Server

# Adding file server mount points

Puppet Server includes a file server for transferring static file content to agents. If you need to serve large files that you don't want to store in source control or distribute with a module, you can make a custom file server mount point and let Puppet serve those files from another directory.

In Puppet code, you can tell the file server is being used when you see a `file` resource that has a `source => puppet:///...` attribute specified.

To set up a mount point:

1. Choose a directory on disk for the mount point, make sure Puppet Server can access it, and add your files to the directory.
2. Edit `fileserver.conf` on your Puppet Server node, so Puppet knows which directory to associate with the new mount point.

3. (Optional) If you want to restrict which nodes can access this mount point, adjust access settings in the `auth.conf` file.

After the mount point is set up, Puppet cod can reference the files you added to the directory at `puppet:///<MOUNT POINT>/<PATH>`.

### Mount points in the Puppet URI

Puppet URIs look like this: `puppet://<SERVER>/<MOUNT POINT>/<PATH>`.

The `<SERVER>` is optional, so it common practice to use `puppet:///` URIs with three slashes. Usually, there is no reason to specify the server. For Puppet agent, `<SERVER>` defaults to the value of the server setting. For Puppet apply, `<SERVER>` defaults to a special mock server with a modules mount point.

`<MOUNT POINT>` is a unique identifier for some collection of files. There are different kinds of mount points:

- Custom mount points correspond to a directory that you specify.
- The `task` mount point works in a similar way to the `modules` mount point but for files that live under the modules `tasks` directory, rather than the `files` directory.
- The special `modules` mount point serves files from the `files` directory of every module. It behaves as if someone had copied the `files` directory from every module into one big directory, renaming each of them with the name of their module. For example, the files in `apache/files/...` are available at `puppet:///modules/apache/...`.
- The special `plugins` mount point serves files from the `lib` directory of every module. It behaves as if someone had copied the contents of every `lib` directory into one big directory, with no additional namespacing. Puppet agent uses this mount point when syncing plugins before a run, but there's no reason to use it in a `file` resource.
- The special `pluginfacts` mount point serves files from the `facts.d` directory of every module to support external facts. It behaves like the `plugins` mount point, but with a different source directory.
- The special `locales` mount point serves files from the `locales` directory of every module to support automatic downloading of module translations to agents. It also behaves like the `plugins` mount point, and also has a different source directory.

`<PATH>` is the remainder of the path to the file, starting from the directory (or imaginary directory) that corresponds to the mount point.

### Creating a new mount point in `fileserver.conf`

The `fileserver.conf` file uses the following syntax to define mount points:

```
[<NAME OF MOUNT POINT>]
    path <PATH TO DIRECTORY>
           allow *
```

In the following example, a file at `/etc/puppetlabs/puppet/installer_files/oracle.pkg` would be available in manifests as `puppet:///installer_files/oracle.pkg`:

```
[installer_files]
    path /etc/puppetlabs/puppet/installer_files
    allow
```

Make sure that the `puppet` user has the right permissions to access that directory and its contents.

Always include the `allow *` line, because the default behavior is to deny all access. To change access to a custom mount point, update the rules in `auth.conf`, as described below. Putting authorization rules in `fileserver.conf` is deprecated.

**CAUTION:** Always restrict write access to mounted directories. The file server follows any symlinks in a file server mount, including links to files that agent nodes cannot access (such as SSL keys). When following symlinks, the file server can access any files readable by Puppet Server's user account.

### Controlling access to a custom mount point in `auth.conf`

By default, any node with a valid certificate can access the files in your new mount point. If a node can fetch a catalog, it can fetch files. If the node can't fetch a catalog, it can't fetch files. This is the same behavior as the special `modules` and `plugins` mount points. If necessary, you can restrict access to a custom mount point in `auth.conf`.

Both the location of `auth.conf`, and the process for editing `auth.conf` differ depending on whether you're using the new Puppet Server authentication configuration, or the legacy configuration.

If you're using the new configuration, and you've disabled the legacy `auth.conf` file by setting `jruby-puppet.use-legacy-auth-conf: false`, you can add a rule to Puppet Server's HOCON-format `auth.conf file`, located at `/etc/puppetlabs/puppetserver/conf.d/auth.conf`.

Your new auth rule must meet the following requirements:

- It must match requests to all four of these prefixes:
  - `/puppet/v3/file_metadata/<MOUNT POINT>`
  - `/puppet/v3/file_metadatas/<MOUNT POINT>`
  - `/puppet/v3/file_content/<MOUNT POINT>`
  - `/puppet/v3/file_contents/<MOUNT POINT>`
- Its `sort-order` must be lower than 500, so that it overrides the default rule for the file server.

For example:

```
{
    # Allow limited access to files in /etc/puppetlabs/puppet/
installer_files:
    match-request: {
        path: "^/puppet/v3/file_(content|metadata)s?/installer_files"
        type: regex
    }
    allow: "*.dev.example.com"
    sort-order: 400
    name: "dev.example.com large installer files"
},
```

If you haven't disabled the legacy `auth.conf` file, add a rule to `/etc/puppetlabs/puppet/auth.conf`.

Your new auth rule must meet the following requirements:

- It must match requests to all four of these prefixes:
  - `/puppet/v3/file_metadata/<MOUNT POINT>`
  - `/puppet/v3/file_metadatas/<MOUNT POINT>`
  - `/puppet/v3/file_content/<MOUNT POINT>`
  - `/puppet/v3/file_contents/<MOUNT POINT>`
- It must be located earlier in the `auth.conf` file than the default `/puppet/v3/file` rule.

For example:

```
# Allow limited access to files in /etc/puppetlabs/puppet/installer_files:
path ~ ^/file_(metadata|content)s?/installer_files/
auth yes
allow *.dev.example.com
           allow_ip 192.168.100.0/24
```

Related topics: Module fundamentals, fileserver.conf: Custom fileserver mount points, Puppet Server configuration files: puppetserver.conf, Puppet Server configuration files: auth.conf.

## Checking the values of settings

Puppet settings are highly dynamic, and their values can come from several different places. To see the actual settings values that a Puppet service uses, run the `puppet config print` command.

### General usage

The `puppet config print` command loads and evaluates settings, and can imitate any of Puppet's other commands and services when doing so. The `--section` and `--environment` options let you control how settings are loaded; for details, see the sections below on imitating different services.

**Note:** To ensure that you're seeing the values Puppet use when running as a service, be sure to use sudo or run the command as `root` or `Administrator`. If you run `puppet config print` as some other user, Puppet might not use the system config file.

To see the value of one setting:

```
sudo puppet config print <SETTING NAME> [--section <CONFIG SECTION>] [--
environment <ENVIRONMENT>]
```

This displays just the value of `<SETTING NAME>`.

To see the value of multiple settings:

```
sudo puppet config print <SETTING 1> <SETTING 2> [...] [--section <CONFIG
 SECTION>] [--environment <ENVIRONMENT>]
```

This displays `name = value` pairs for all requested settings.

To see the value of all settings:

```
sudo puppet config print [--section <CONFIG SECTION>] [--environment
 <ENVIRONMENT>]
```

This displays `name = value` pairs for all settings.

### Config sections

The `--section` option specifies which section of `puppet.conf` to use when finding settings. It is optional, and defaults to `main`. Valid sections are:

- `main` (default) — used by all commands and services
- `master` — used by the Puppet master service and the `puppetserver ca` command
- `agent` — used by the Puppet agent service
- `user` — used by the Puppet apply command and most other commands

As usual, the other sections override the `main` section if they contain a setting; if they don't, the value from `main` is used, or a default value if the setting isn't present there.

### Environments

The `--environment` option specifies which environment to use when finding settings. It is optional and defaults to the value of the `environment` setting in the `user` section (usually `production`, because it's rare to specify an environment in `user`).

You can only specify environments that exist.

This option is primarily useful when looking up settings used by the master service, because it's rare to use environment config sections for Puppet apply and Puppet agent.

#### Imitating Puppet master and `puppetserver ca`

To see the settings the Puppet master service and the `puppetserver ca` command would use:

- Specify `--section master`.
- Use the `--environment` option to specify the environment you want settings for, or let it default to `production`.
- Remember to use `sudo`.
- If your Puppet master is managed as a Rack application (for example, with Passenger), check the `config.ru` file to make sure it's using the confdir and vardir that you expect. If it's using non-standard ones, you need to specify them on the command line with the `--confdir` and `--vardir` options; otherwise you might not see the correct values for settings.

To see the effective modulepath used in the `dev` environment:

```
sudo puppet config print modulepath --section master --environment dev
```

This returns something like:

```
/etc/puppetlabs/code/environments/dev/modules:/etc/puppetlabs/code/modules:/
opt/puppetlabs/puppet/modules
```

To see whether PuppetDB is configured for exported resources:

```
sudo puppet config print storeconfigs storeconfigs_backend --section master
```

This returns something like:

```
storeconfigs = true
storeconfigs_backend = puppetdb
```

#### Imitating Puppet agent

To see the settings the Puppet agent service would use:

- Specify `--section agent`.
- Remember to use `sudo`.
- If you are seeing something unexpected, check your Puppet agent init script or cron job to make sure it is using the standard confdir and vardir, is running as root, and isn't overriding other settings with command line options. If it's doing anything unusual, you might have to set more options for the `config print` command.

To see whether the agent is configured to use manifest ordering when applying the catalog:

```
sudo puppet config print ordering --section agent
```

This returns something like:

```
manifest
```

#### Imitating `puppet apply`

To see the settings the Puppet apply command would use:

- Specify `--section` user.
- Remember to use `sudo`.
- If you are seeing something unexpected, check the cron job or script that is responsible for configuring the machine with Puppet apply. Make sure it is using the standard confdir and vardir, is running as root, and isn't overriding other settings with command line options. If it's doing anything unusual, you might have to set more options for the `config print` command.

To see whether Puppet apply is configured to use reports:

```
sudo puppet config print report reports --section user
```

This returns something like:

```
report = true
reports = store,http
```

## Editing settings on the command line

Puppet loads most of its settings from the `puppet.conf` config file. You can edit this file directly, or you can change individual settings with the `puppet config set` command.

Use `puppet config set` for:

- Fast one-off config changes,
- Scriptable config changes in provisioning tools,

If you find yourself changing many settings, edit the `puppet.conf` file instead, or manage it with a template.

### Usage

To assign a new value to a setting, run:

```
sudo puppet config set <SETTING NAME> <VALUE> --section <CONFIG SECTION>
```

This declaratively sets the value of `<SETTING NAME>` to `<VALUE>` in the specified config section, regardless of whether the setting already had a value.

### Config sections

The `--section` option specifies which section of `puppet.conf` to modify. It is optional, and defaults to `main`. Valid sections are:

- `main` (default) — used by all commands and services
- `master` — used by the Puppet master service and the `puppetserver ca` command
- `agent` — used by the Puppet agent service
- `user` — used by the `puppet apply` command and most other commands

When modifying the system config file, use `sudo` or run the command as `root` or Administrator.

### Example

Consider the following `puppet.conf` file:

```
[main]
certname = agent01.example.com
server = master.example.com
vardir = /var/opt/lib/pe-puppet

[agent]
report = true
graph = true
pluginsync = true

[master]
dns_alt_names = master,master.example.com,puppet,puppet.example.com
```

If you run the following commands:

```
sudo puppet config set reports puppetdb --section master
sudo puppet config set ordering manifest
```

The `puppet.conf` file now looks like this:

```
[main]
certname = agent01.example.com
server = master.example.com
vardir = /var/opt/lib/pe-puppet
ordering = manifest

[agent]
report = true
graph = true
pluginsync = true

[master]
dns_alt_names = master,master.example.com,puppet,puppet.example.com
reports = puppetdb
```

# Important directories and files

Puppet consists of a number of directories and files, and each one has an important role ranging from Puppet code storage and configuration files to manifests and module paths.

## Code and data directory (codedir)

The codedir is the main directory for Puppet code and data. It is used by Puppet master and Puppet apply, but not by Puppet agent. It contains environments (which contain your manifests and modules), a global modules directory for all environments, and your Hiera data and configuration.

### Location

The codedir is located in one of the following locations:

- *nix: `/etc/puppetlabs/code`
- *nix non-root users: `~/.puppetlabs/etc/code`
- Windows: `%PROGRAMDATA%\PuppetLabs\code` (usually `C:\ProgramData\PuppetLabs\code`)

When Puppet is running as root, as a Windows user with administrator privileges, or as the `puppet` user, it uses a system-wide codedir. When running as a non-root user, it uses a codedir in that user's home directory.

When running Puppet commands and services as `root` or `puppet`, use the system codedir. To use the same codedir as the Puppet agent, or Puppet master, run admin commands such as `puppet module` with `sudo`.

To configure the location of the codedir, set the `codedir` setting in your `puppet.conf` file, such as:

```
codedir = /etc/puppetlabs/code
```

**Important:** Puppet Server doesn't use the codedir setting in `puppet.conf`, and instead uses the `jruby-puppet.master-code-dir` setting in `puppetserver.conf`. When using a non-default codedir, you must change both settings.

### Interpolation of `$codedir`

The value of the codedir is discovered before other settings, so you can refer to it in other `puppet.conf` settings by using the `$codedir` variable in the value. For example, the `$codedir` variable is used as part of the value for the `environmentpath` setting:

```
[master]
   environmentpath = $codedir/override_environments:$codedir/environments
```

This allows you to avoid absolute paths in your settings and keep your Puppet-related files together.

### Contents

The codedir contains environments, including manifests and modules, a global modules directory for all environments, and Hiera data.

The code and data directories are:

- `environments` : Contains alternate versions of the `modules` and `manifests` directories, to enable code changes to be tested on smaller sets of nodes before entering production.
- `modules` : The main directory for modules.

## Config directory (confdir)

Puppet's `confdir` is the main directory for the Puppet configuration. It contains configuration files and the SSL data.

### Location

The confdir is located in one of the following locations:

- *nix root users: `/etc/puppetlabs/puppet`
- Non-root users: `~/.puppetlabs/etc/puppet`
- Windows: `%PROGRAMDATA%\PuppetLabs\puppet\etc` (usually `C:\ProgramData\PuppetLabs\puppet\etc`)

When Puppet is running as `root`, a Windows user with administrator privileges, or the `puppet` user, it uses a system-wide confdir. When running as a non-root user, it uses a confdir in that user's home directory.

When running Puppet commands and services as `root` or `puppet`, usually you want to use the system codedir. To use the same codedir as the Puppet agent or Puppet master, run admin commands with `sudo`.

Puppet's confdir can't be set in the `puppet.conf`, because Puppet needs the confdir to locate that config file. Instead, run commands with the `--confdir` parameter to specify the confdir. If `--confdir` isn't specified when a Puppet application is started, the command uses the default confdir location.

Puppet Server uses the `jruby-puppet.master-conf-dir` setting in `puppetserver.conf` to configure its confdir. If you are using a non-default confdir, you must specify `--confdir` when you run commands like `puppet module` to ensure they use the same directories as Puppet Server.

### Interpolation of `$confdir`

The value of the confdir is discovered before other settings, so you can reference it, using the `$confdir` variable, in the value of any other setting in `puppet.conf`.

If you need to set nonstandard values for some settings, using the `$confdir` variable allows you to avoid absolute paths and keep your Puppet-related files together.

### Contents

The confdir contains several config files and the SSL data. You can change their locations, but unless you have a technical reason that prevents it, use the default structure. Click the links to see documentation for the files and directories in the codedir.

On all nodes, agent and master, the confdir contains the following directories and config files:

- `ssl` directory: contains each node's certificate infrastructure.
- `puppet.conf`: Puppet's main config file.
- `csr_attributes.yaml`: Optional data to be inserted into new certificate requests.

On master nodes, and sometimes standalone nodes that run Puppet apply, the confdir also contains:

- `auth.conf`: Access control rules for the master's network services.
- `fileserver.conf`: Configuration for additional fileserver mount points.
- `hiera.yaml`: The global configuration for Hiera data lookup. Environments and modules can also have their own `hiera.yaml` files.

  **Note:** To provide backward compatibility for some existing Puppet 4 installations, if a `hiera.yaml` file exists in the codedir, it takes precedence over `hiera.yaml` in the confdir. To ensure that Puppet honors the configuration in the confdir, remove any `hiera.yaml` file that is present in the codedir.
- `routes.yaml` : Advanced configuration of indirector behavior.

On certificate authority masters, the confdir also contains:

- `autosign.conf` : List of pre-approved certificate requests.

On nodes that are acting as a proxy for configuring network devices, the confdir also contains:

- `device.conf`: Configuration for network devices managed by the `puppet device` command.

## Main manifest directory

Puppet starts compiling a catalog either with a single manifest file or with a directory of manifests that are treated like a single file. This starting point is called the *main manifest* or *site manifest*.

For more information about how the site manifest is used in catalog compilation, see Catalog compilation.

### Specifying the manifest for Puppet apply

The `puppet apply` command uses the manifest you pass to it as an argument on the command line:

```
puppet apply /etc/puppetlabs/code/environments/production/manifests/site.pp
```

You can pass Puppet apply either a single `.pp` file or a directory of `.pp` files. Puppet apply uses the manifest you pass it, not an environment's manifest.

### Specifying the manifest for Puppet master

Puppet master uses the main manifest set by the current node's environment, whether that manifest is a single file or a directory of `.pp` files.

By default, the main manifest for an environment is `<ENVIRONMENTS DIRECTORY>/<ENVIRONMENT>/manifests`, for example `/etc/puppetlabs/code/environments/production/manifests`. You can configure the manifest per-environment, and you can also configure the default for all environments.

To determine its main manifest, an environment uses the `manifest` setting in `environment.conf`. This can be an absolute path or a path relative to the environment's main directory.

If the `environment.confmanifest` setting is absent, it uses the value of the `default_manifest` setting from the `puppet.conf` file. The `default_manifest` setting defaults to `./manifests`. Similar to the environment's `manifest` setting, the value of `default_manifest` can be an absolute path or a path relative to the environment's main directory.

To force all environments to ignore their own `manifest` setting and use the `default_manifest` setting instead, set `disable_per_environment_manifest = true` in `puppet.conf`.

To check which manifest your Puppet master uses for a given environment, run:

```
puppet config print manifest --section master --environment <ENVIRONMENT>
```

For more information, see Creating environments, and Checking values of configuration settings.

### Manifest directory behavior

When the main manifest is a directory, Puppet parses every .pp file in the directory in alphabetical order and evaluates the combined manifest. It descends into all subdirectories of the manifest directory and loads files in depth-first order. For example, if the manifest directory contains a directory named 01, and a file named 02.pp, it parses the files in 01 before it parses 02.pp.

Puppet treats the directory as one manifest, so, for example, a variable assigned in the file 01_all_nodes.pp is accessible in node_web01.pp.

## The modulepath

The master service and the puppet apply command load most of their content from modules found in one or more directories. The list of directories where Puppet looks for modules is called the *modulepath*. The modulepath is set by the current node's environment.

The modulepath is an ordered list of directories, with earlier directories having priority over later ones. Use the system path separator character to separate the directories in the modulepath list. On *nix systems, use a colon (:); on Windows use a semi-colon (;).

For example, on *nix:

```
/etc/puppetlabs/code/environments/production/modules:/etc/puppetlabs/code/
modules:/opt/puppetlabs/puppet/modules
```

On Windows:

```
C:/ProgramData/PuppetLabs/code/environments/production/modules;C:/
ProgramData/PuppetLabs/code/modules
```

Each directory in the modulepath must contain only valid Puppet modules, and the names of those modules must follow the modules naming rules. Dashes and periods in module names cause errors. For more information, see Modules fundamentals.

By default, the modulepath is set by the current node's environment in environment.conf. For example, using *nix paths:

```
modulepath = site:dist:modules:$basemodulepath
```

To see what the modulepath is for an environment, run:

```
sudo puppet config print modulepath --section master --environment
 <ENVIRONMENT_NAME>
```

For more information about environments, see Environments.

### Setting the modulepath and base modulepath

Each environment sets its full modulepath in the environment.conf file with the modulepath setting. The modulepath setting can only be set in environment.conf. It configures the entire modulepath for that environment.

The modulepath can include relative paths, such as ./modules or ./site. Puppet looks for these paths inside the environment’s directory.

The default modulepath value for an environment is the environment’s modules directory, plus the base modulepath. On *nix, this is ./modules:$basemodulepath.

The **base modulepath** is a list of global module directories for use with all environments. You can configure it with the `basemodulepath` setting in the `puppet.conf` file, but its default value is probably suitable. The default on *nix:

```
$codedir/modules:/opt/puppetlabs/puppet/modules
```

On Windows:

```
$codedir\modules
```

If you want an environment to have access to the global module directories, include `$basemodulepath` in the environment's modulepath setting:

```
modulepath = site:dist:modules:$basemodulepath
```

#### Using the `--modulepath` option with Puppet apply

When running Puppet apply on the command line, you can optionally specify a modulepath with the `--modulepath` option, which overrides the modulepath from the current environment.

#### Absent, duplicate, and conflicting content from modules

Puppet uses modules it finds in every directory in the modulepath. Directories in the modulepath can be empty or absent. This is not an error; Puppet does not attempt to load modules from those directories. If no modules are present across the entire modulepath, or if modules are present but none of them contains a `lib` directory, the agent logs an error when attempting to sync plugins from the master. This error is benign and doesn't prevent the rest of the run.

If the modulepath contains multiple modules with the same name, Puppet uses the version from the directory that comes **earliest** in the modulepath. Modules in directories earlier in the modulepath override those in later directories.

For most content, this earliest-module-wins behavior is on an all-or-nothing, **per-module** basis — all of the manifests, files, and templates in the first-encountered version are available for use, and none of the content from any subsequent versions is available. This behavior covers:

- Puppet code (from `manifests`).
- Files (from `files`).
- Templates (from `templates`).
- External facts (from `facts.d`).
- Ruby plugins synced to agent nodes (from `lib`).

**CAUTION:** Puppet sometimes displays unexpected behavior with Ruby plugins that are loaded directly from modules. This includes:

- Plugins used by the master (custom resource types, custom functions).
- Plugins used by `puppet apply`.
- Plugins present in the agent's modulepath (which is usually empty but night not be when running an agent on a node that is also a master).

In this case, the plugins are handled on a **per-file** basis instead of per-module. If a duplicate module in an later directory has additional plugin files that don't exist in the first instance of the module, those extra files *are* loaded, and Puppet uses a a mixture of files from both versions of the module.

If you refactor a module's Ruby plugins, and maintain two versions of that module in your modulepath, it can have unexpected results.

This is a byproduct of how Ruby works and is not intentional or controllable by Puppet; a fix is not expected.

## SSL directory (ssldir)

Puppet stores its certificate infrastructure in the SSL directory (ssldir) which has a similar structure on all Puppet nodes, whether they are agent nodes, Puppet masters, or the certificate authority (CA) master.

### Location

By default, the ssldir is a subdirectory of the confdir.

You can change its location using the `ssldir` setting in the `puppet.conf` file. See the Configuration reference for more information.

**Note:** The content of the ssldir is generated, grows over time, and is relatively difficult to replace. Some third-party Puppet packages for Linux place the ssldir in the cache directory (vardir) instead of the confdir. When a distro changes the ssldir location, it sets `ssldir` in the `$confdir/puppet.conf` file, usually in the `[main]` section.

To see the location of the ssldir on one of your nodes, run: `puppet config print ssldir`

### Contents

The ssldir contains Puppet certificates, private keys, certificate signing requests (CSRs), and other cryptographic documents.

The ssldir on an agent or master contains:

- A private key: `private_keys/<certname>.pem`
- A signed certificate: `certs/<certname>.pem`
- A copy of the CA certificate: `certs/ca.pem`
- A copy of the certificate revocation list (CRL): `crl.pem`
- A copy of its sent CSR: `certificate_requests/<certname>.pem`

**Tip:** Puppet does not save its public key to disk, because the public key is derivable from its private key and is contained in its certificate. If you need to extract the public key, use `$ openssl rsa -in $(puppet config print hostprivkey) -pubout`

If these files don’t exist on a node, it's because they are generated locally or requested from the CA Puppet master.

Agent and master credentials are identified by certname, so an agent process and a master process running on the same server can use the same credentials.

The ssldir for the Puppet CA, which runs on the CA master, contains similar credentials: private and public keys, a certificate, and a master copy of the CRL. It maintains a list of all signed certificates in the deployment, a copy of each signed certificate, and an incrementing serial number for new certificates. To keep it separated from general Puppet credentials on the same server, all of the CA’s data is stored in the `ca` subdirectory.

### The ssldir directory structure

All of the files and directories in the `ssldir` directory have corresponding Puppet settings, which can be used to change their locations. Generally, though, don't change the default values unless you have a specific problem to work around.

Ensure the permissions mode of the ssldir is 0771. The directory and each file in it is owned by the user that Puppet runs as: root or Administrator on agents, and defaulting to `puppet` or `pe-puppet` on a master. Set up automated management for ownership and permissions on the ssldir.

The ssldir has the following structure. See the Configuration reference for details about each `puppet.conf` setting listed:

- `ca` directory (on the CA master only): Contains the files used by Puppet's certificate authority. Mode: 0755. Setting: `cadir`.
  - `ca_crl.pem`: The master copy of the certificate revocation list (CRL) managed by the CA. Mode: 0644. Setting: `cacrl`.
  - `ca_crt.pem`: The CA's self-signed certificate. This cannot be used as a master or agent certificate; it can only be used to sign certificates. Mode: 0644. Setting: `cacert`.
  - `ca_key.pem`: The CA's private key, and one of the most security-critical files in the Puppet certificate infrastructure. Mode: 0640. Setting: `cakey`.
  - `ca_pub.pem`: The CA's public key. Mode: 0644. Setting: `capub`.
  - `inventory.txt`: A list of the certificates the CA signed, along with their serial numbers and validity periods. Mode: 0644. Setting: `cert_inventory`.
  - `requests` (directory): Contains the certificate signing requests (CSRs) that have been received but not yet signed. The CA deletes CSRs from this directory after signing them. Mode: 0755. Setting: `csrdir`.
    - `<name>.pem`: CSR files awaiting signing.
  - `serial`: A file containing the serial number for the next certificate the CA signs. This is incremented with each new certificate signed. Mode: 0644. Setting: `serial`.
  - `signed` (directory): Contains copies of all certificates the CA has signed. Mode: 0755. Setting: `signeddir`.
    - `<name>.pem`: Signed certificate files.
- `certificate_requests` (directory): Contains CSRs generated by this node in preparation for submission to the CA. CSRs stay in this directory even after they have been submitted and signed. Mode: 0755. Setting: `requestdir`.
  - `<certname>.pem`: This node's CSR. Mode: 0644. Setting: `hostcsr`.
- `certs` (directory): Contains signed certificates present on the node. This includes the node's own certificate, and a copy of the CA certificate for validating certificates presented by other nodes. Mode: 0755. Setting: `certdir`.
  - `<certname>.pem`: This node's certificate. Mode: 0644. Setting: `hostcert`.
  - `ca.pem`: A local copy of the CA certificate. Mode: 0644. Setting: `localcacert`.
- `crl.pem`: A copy of the certificate revocation list (CRL) retrieved from the CA, for use by agents or masters. Mode: 0644. Setting: `hostcrl`.
- `private` (directory): Usually, does not contain any files. Mode: 0750. Setting: `privatedir`.
  - `password`: The password to a node's private key. Usually not present. The conditions in which this file would exist are not defined. Mode: 0640. Setting: `passfile`.
- `private_keys` (directory): Contains the node's private key and, on the CA, private keys created by the `puppet cert generate` command. It never contains the private key for the CA certificate. Mode: 0750. Setting: `privatekeydir`.
  - `<certname>.pem`: This node's private key. Mode: 0600. Setting: `hostprivkey`.
- `public_keys` (directory): Contains public keys generated by this node in preparation for generating a CSR. Mode: 0755. Setting: `publickeydir`.
  - `<certname>.pem`: This node's public key. Mode: 0644. Setting: `hostpubkey`.

## Cache directory (vardir)

As part of its normal operations, Puppet generates data which is stored in a cache directory called vardir. You can mine the data in vardir for analysis, or use it to integrate other tools with Puppet.

### Location

The cache directory for Puppet Server defaults to `/opt/puppetlabs/server/data/puppetserver`.

The cache directory for the Puppet agent and Puppet apply can be found at one of the following locations:

- *nix systems: `/opt/puppetlabs/puppet/cache`.

- Non-root users: `~/.puppetlabs/opt/puppet/cache`.
- Windows: `%PROGRAMDATA%\PuppetLabs\puppet\cache` (usually `C:\Program Data\PuppetLabs\puppet\cache`).

When Puppet is running as root, a Windows user with administrator privileges, or the `puppet` user, uses a system-wide cache directory. When running as a non-root user, it uses a cache directory in the user's home directory.

Because you usually run Puppet's commands and services as root or `puppet`, the system cache directory is what you usually want to use.

**Important:** To use the same directories as the agent or master, admin commands like `puppet cert`, must run with `sudo`.

**Note:** When Puppet master is running as a Rack application, the `config.ru` file must explicitly set `--vardir` to the system cache directory. The example `config.ru` file provided with the Puppet source does this.

You can specify Puppet's cache directory on the command line by using the `--vardir` option, but you can't set it in `puppet.conf`. If `--vardir` isn't specified when a Puppet application is started, it uses the default cache directory location.

To configure the Puppet Server cache directory, use the `jruby-puppet.master-var-dir` setting in `puppetserver.conf` .

### Interpolation of `$vardir`

The value of the vardir is discovered before other settings, so you can reference it using the `$vardir` variable in the value of any other setting in `puppet.conf` or on the command line.

For example:

```
[main]
  ssldir = $vardir/ssl
```

If you need to set nonstandard values for some settings, using the `$vardir` variable allows you to avoid absolute paths and keep your Puppet-related files together.

### Contents

The vardir contains several subdirectories. Most of these subdirectories contain a variable amount of generated data, some contain notable individual files, and some directories are used only by agent or master processes.

To change the locations of specific vardir files and directories, edit the settings in `puppet.conf`. For more information about each item below, see the Configuration reference.

| Directory name | Config setting | Notes |
|---|---|---|
| `bucket` | `bucketdir` | |
| `client_data` | `client_datadir` | |
| `clientbucket` | `clientbucketdir` | |
| `client_yaml` | `clientyamldir` | |
| `devices` | `devicedir` | |
| `lib/facter` | `factpath` | |
| `facts` | `factpath` | |
| `facts.d` | `pluginfactdest` | |

| Directory name | Config setting | Notes |
| --- | --- | --- |
| lib | libdir, plugindest | Puppet uses this as a cache for plugins (custom facts, types and providers, functions) synced from a Puppet master. Do not change its contents. If you delete it, the plugins are restored on the next Puppet run. |
| puppet-module | module_working_dir | |
| puppet-module/skeleton | module_skeleton_dir | |
| reports | reportdir | When the option to store reports is enabled, a master stores reports received from agents as YAML files in this directory. You can mine these reports for analysis. |
| server_data | serverdatadir | |
| state | statedir | See table below for more details about the state directory contents. |
| yaml | yamldir | |

The state directory contains the following files and directories:

| File or directory name | Config setting | Notes |
| --- | --- | --- |
| agent_catalog_run.lock | agent_catalog_run_lockfile | |
| agent_disabled.lock | agent_disabled_lockfile | |
| classes.txt | classfile | This file is useful for external integration. It lists all of the classes assigned to this agent node. |
| graphs directory | graphdir | When graphing is enabled, agent nodes write a set of .dot graph files to this directory. Use these graphs to diagnose problems with the catalog application, or visualizing the configuration catalog. |
| last_run_summary.yaml | lastrunfile | |
| last_run_report.yaml | lastrunreport | |
| resources.txt | resourcefile | |
| state.yaml | statefile | |

# Environments

Environments are isolated groups of agent nodes.



## About environments

An environment is a branch that gets turned into a directory on your master.

A master serves each environment with its own main manifest and module path. This lets you use different versions of the same modules for different groups of nodes, which is useful for testing changes to your code before implementing them on production machines.

Related topics: main manifests, module paths.

### Access environment name in manifests

If you want to share code across environments, use the `$environment` variable in your manifests.

To get the name of the current environment:

1. Use the `$environment` variable, which is set by the master.

### Environment scenarios

The main uses for environments fall into three categories: permanent test environments, temporary test environments, and divided infrastructure.

#### Permanent test environments

In a permanent test environment, there is a stable group of test nodes where all changes must succeed before they can be merged into the production code. The test nodes are a smaller version of the whole production infrastructure. They are either short-lived cloud instances or longer-lived virtual machines (VMs) in a private cloud. These nodes stay in the test environment for their whole lifespan.

#### Temporary test environments

In a temporary test environment, you can test a single change or group of changes by checking the changes out of version control into the `$codedir/environments` directory, where it is detected as a new environment. A temporary test environment can either have a descriptive name or use the commit ID from the version that it is based on. Temporary environments are good for testing individual changes, especially if you need to iterate quickly while developing them. When you’re done with a temporary environment, you can delete it. The nodes in a temporary environment are short-lived cloud instances or VMs, which are destroyed when the environment ends.

#### Divided infrastructure

If parts of your infrastructure are managed by different teams that do not need to coordinate their code, you can split them into environments.

## Environments limitations

Environments have limitations, including leakage and conflicts with exported resources.

### Plugins can leak between environments

Environment leakage occurs when different versions of Ruby files, such as resource types, exist in multiple environments. When these files are loaded on the master, the first version loaded is treated as global. Subsequent requests in other environments get that first loaded version. Environment leakage does not affect the agent, as agents are only in one environment at any given time. For more information, see below for troubleshooting environment leakage.

### Exported resources can conflict or cross over

Nodes in one environment can collect resources that were exported from another environment, which causes problems — either a compilation error due to identically titled resources, or creation and management of unintended resources. The solution is to run separate masters for each environment if you use exported resources.

## Troubleshoot environment leakage

Environment leakage is one of the limitations of environments.

Use one of the following methods to avoid environmental leakage:

- For resource types, you can avoid environment leaks with the the `puppet generate types` command as described in environment isolation documentation. This command generates resource type metadata files to ensure that each environment uses the right version of each type.
- This issue occurs only with the `Puppet::Parser::Functions` API. To fix this, rewrite functions with the modern functions API, which is not affected by environment leakage. You can include helper code in the function definition, but if helper code is more complex, package it as a gem and install for all environments.
- Report processors and indirector termini are still affected by this problem, so put them in your global Ruby directories rather than in your environments. If they are in your environments, you must ensure they all have the same content.

# Creating environments

An environment is a branch that gets turned into a directory on your Puppet master. Environments are turned on by default.

## Environment structure

The structure of an environment follows several conventions.

When you create an environment, you give it the following structure:

- It contains a `modules` directory, which becomes part of the environment's default module path.
- It contains a `manifests` directory, which is the environment's default main manifest.
- If you are using Puppet 5, it can optionally contain a `hiera.yaml` file.
- It can optionally contain an `environment.conf` file, which can locally override configuration settings, including `modulepath` and `manifest`.

  **Note:** Environment names can contain lowercase letters, numbers, and underscores. They must match the following regular expression rule: `\A[a-z0-9_]+\Z`. If you are using Puppet 5, remove the `environment_data_provider` setting.

## Environment resources

An environment specifies resources that the master uses when compiling catalogs for agent nodes. The `modulepath`, the main manifest, Hiera data, and the config version script, can all be specified in `environment.conf`.

### The modulepath

The `modulepath` is the list of directories Puppet loads modules from. By default, Puppet loads modules first from the environment's directory, and second from the master's `puppet.conf` file's `basemodulepath` setting, which can be multiple directories. If the modules directory is empty or absent, Puppet only uses modules from directories in the `basemodulepath`.

Related topics: module path.

### The main manifest

The main manifest is the starting point for compiling a catalog. Unless you say otherwise in `environment.conf`, an environment uses the global `default_manifest` setting to determine its main manifest. The value of this setting can be an absolute path to a manifest that all environments share, or a relative path to a file or directory inside each environment.

The default value of `default_manifest` is `./manifests` — the environment's own manifests directory. If the file or directory specified by `default_manifest` is empty or absent, Puppet does not fall back to any other manifest. Instead, it behaves as if it is using a blank main manifest. If you specify a value for this setting, the global manifest setting from `puppet.conf` is not be used by an environment.

Related topics: main manifest, environment.conf, default_manifest setting, puppet.conf.

### Hiera data

Each environment can use its own Hiera hierarchy and provide its own data.

Related topics: Hiera config file syntax.

### The config version script

Puppet automatically adds a config version to every catalog it compiles, as well as to messages in reports. The version is an arbitrary piece of data that can be used to identify catalogs and events. By default, the config version is the time at which the catalog was compiled (as the number of seconds since January 1, 1970).

### The environment.conf file

An environment can contain an `environment.conf` file, which can override values for certain settings.

The `environment.conf` file overrides these settings:

- `modulepath`
- `manifest`
- `config_version`
- `environment_timeout`

Related topics: environment.conf

## Create an environment

Create an environment by adding a new directory of configuration data.

1. Inside your code directory, create a directory called `environments`.
2. Inside the `environments` directory, create a directory with the name of your new environment using the structure: `$codedir/environments/`
3. Create a `modules` directory and a `manifests` directory. These two directories contain your Puppet code.

4. Configure a `modulepath`:
   a) Set `modulepath` in its `environment.conf` file . If you set a value for this setting, the global `modulepath` setting from `puppet.conf` is not used by an environment.
   b) Check the `modulepath` by specifying the environment when requesting the setting value:

```
$ sudo puppet config print modulepath --section master --environment
 test /etc/puppetlabs/code/environments/test/modules:/etc/puppetlabs/code/
modules:/opt/puppetlabs/puppet/modules.
```

   **Note:** In Puppet Enterprise (PE), every environment must include `/opt/puppetlabs/puppet/modules` in its `modulepath`, because PE uses modules in that directory to configure its own infrastructure.
5. Configure a main manifest:
   a) Set manifest in its `environment.conf` file. As with the global `default_manifest` setting, you can specify a relative path (to be resolved within the environment's directory) or an absolute path.
   b) Lock all environments to a single global manifest with the `disable_per_environment_manifest` setting — preventing any environment setting its own main manifest.
6. To specify an executable script that determines an environment's config version:
   a) Specify a path to the script in the `config_version` setting in its `environment.conf` file. Puppet runs this script when compiling a catalog for a node in the environment, and uses its output as the config version. If you specify a value here, the global `config_version` setting from `puppet.conf` is not used by an environment.

   **Note:** If you're using a system binary like git `rev-parse`, specify the absolute path to it. If `config_version` is set to a relative path, Puppet looks for the binary in the environment, not in the system's PATH.

Related topics: Deploying environments with r10k, Code Manager control repositories, disable_per_environment_manifest

## Assign nodes to environments via an ENC

You can assign agent nodes to environments by using an external node classifier (ENC). By default, all nodes are assigned to a default environment named production.

The interface to set the environment for a node is different for each ENC. Some ENCs cannot manage environments. When writing an ENC:

1. Ensure that the environment key is set in the YAML output that the ENC returns. If the environment key isn't set in the ENC's YAML output, the master uses the environment requested by the agent.

**Note:** The value from the ENC is authoritative, if it exists. If the ENC doesn't specify an environment, the node's config value is used.

Related topics: writing ENCs.

## Assign nodes to environments via the agent's config file

You can assign agent nodes to environments by using the agent's config file. By default, all nodes are assigned to a default environment named production.

To configure an agent to use an environment:

1. Open the agent's `puppet.conf` file in an editor.
2. Find the `environment` setting in either the agent or main section.
3. Set the value of the `environment` setting to the name of the environment you want the agent to be assigned to.

When that node requests a catalog from the master, it requests that environment. If you are using an ENC and it specifies an environment for that node, it overrides whatever is in the config file.

**Note:** Nodes cannot be assigned to unconfigured environments. If a node is assigned to an environment that does not exist — no directory of that name in any of the environment path directories — the master fails to compile its catalog.

The one exception to this is if the default production environment does not exist. In this case, the agent successfully retrieves an empty catalog.

## Global settings for configuring environments

The settings in the master's `puppet.conf` file configure how Puppet finds and uses environments.

### environmentpath

The `environmentpath` setting is the list of directories where Puppet looks for environments. The default value for `environmentpath` is `$codedir/environments`. If you have more than one directory, separate them by colons and put them in order of precedence.

In this example, `temp_environments` is searched before `environments`:

```
$codedir/temp_environments:$codedir/environments
```

If environments with the same name exist in both paths, Puppet uses the first environment with that name that it encounters.

Put the `environmentpath` setting in the main section of the `puppet.conf` file.

### basemodulepath

The `basemodulepath` setting lists directories of global modules that all environments can access by default. Some modules can be made available to all environments. The `basemodulepath` setting configures the global module directories.

By default, it includes `$codedir/modules` for user-accessible modules and `/opt/puppetlabs/puppet/modules` for system modules.

Add additional directories containing global modules by setting your own value for `basemodulepath`.

Related topics: modulepath.

### environment_timeout

The `environment_timeout` setting sets how often the master refreshes information about environments. It can be overridden per-environment.

This setting defaults to 0 (caching disabled), which lowers the performance of your master but makes it easy for new users to deploy updated Puppet code. After your code deployment process is mature, change this setting to unlimited.

### disable_per_environment_manifest

The `disable_per_environment_manifest` setting lets you specify that all environments use a shared main manifest.

When `disable_per_environment_manifest` is set to true, Puppet uses the same global manifest for every environment. If an environment specifies a different manifest in `environment.conf`, Puppet does not compile catalogs nodes in that environment, to avoid serving catalogs with potentially wrong contents.

If this setting is set to true, the `default_manifest` value must be an absolute path.

### default_manifest

The `default_manifest` setting specifies the main manifest for any environment that doesn't set a manifest value in `environment.conf`. The default value of `default_manifest` is `./manifests` — the environment's own manifests directory.

The value of this setting can be:

- An absolute path to one manifest that all environments share.

- A relative path to a file or directory inside each environment's directory.

Related topics: default_manifest setting.

### Configure the environment timeout setting

The `enviroment_timeout` setting determines how often the Puppet master caches the data it loads from an environment. For best performance, change the settings after you have a mature code deployment process.

1. Set `environment_timeout = unlimited` in `puppet.conf`.
2. Change your code deployment process to refresh the master whenever you deploy updated code. For example, set a postrun command in your r10k config or add a step to your continuous integration job.
   - With Puppet Server, refresh environments by calling the `environment-cache` API endpoint. Ensure you have write access to the puppet-admin section of the `puppetserver.conf` file.
   - With a Rack master, restart the web server or the application server. Passenger lets you touch a `restart.txt` file to refresh an application without restarting Apache. See the Passenger docs for details.

The `environment-timeout` setting can be overridden per-environment in `environment.conf`.

**Note:** Only use the value 0 or unlimited. Most masters use a pool of Ruby interpreters, which all have their own cache timers. When these timers are out of sync, agents can be served inconsistent catalogs. To avoid that inconsistency, refresh the master when deploying.

## Environment isolation

Environment isolation prevents resource types from leaking between your various environments.

If you use multiple environments with Puppet, you might encounter issues with multiple versions of the same resource type leaking between your various environments on the master. This doesn't happen with built-in resource types, but it can happen with any other resource types.

This problem occurs because Ruby resource type bindings are global in the Ruby runtime. The first loaded version of a Ruby resource type takes priority, and then subsequent requests to compile in other environments get that first-loaded version. Environment isolation solves this issue by generating and using metadata that describes the resource type implementation, instead of using the Ruby resource type implementation, when compiling catalogs.

**Note:** Other environment isolation problems, such as external helper logic issues or varying versions of required gems, are not solved by the generated metadata approach. This fixes only resource type leaking. Resource type leaking is a problem that affects only masters, not agents.

### Enable environment isolation with Puppet

To use environment isolation, generate metadata files that Puppet can use instead of the default Ruby resource type implementations.

1. On the command line, run `puppet generate types --environment <ENV_NAME>` for each of your environments. For example, to generate metadata for your production environment, run: `puppet generate types --environment production`
2. Whenever you deploy a new version of Puppet, overwrite previously generated metadata by running `puppet generate types --environment <ENV_NAME> --force`

### Enable environment isolation with r10k

To use environment isolation with r10k, generate types for each environment every time r10k deploys new code.

1. To generate types with r10k, use one of the following methods:
   - Modify your existing r10k hook to run the `generate types` command after code deployment.
   - Create and use a script that first runs r10k for an environment, and then runs `generate types` as a post run command.

2. If you have enabled environment-level purging in r10k, whitelist the `resource_types` folder so that r10k does not purge it.

**Note:** In Puppet Enterprise (PE), environment isolation is provided by Code Manager Environment isolation is not supported for r10k with PE.

## Troubleshoot environment isolation

If the `generate types` command cannot generate certain types, if the type generated has missing or inaccurate information, or if the generation itself has errors or fails, you get a catalog compilation error of "`type not found`" or "`attribute not found`."

1. To fix catalog compilation errors:
   a) Ensure that your Puppet resource types are correctly implemented. In addition to implementation errors, check for types with title patterns that contain a proc or a lambda, as these types cannot be generated. Refactor any problem resource types.
   b) Regenerate the metadata by removing the environment's `.resource_types` directory and running the `generate types` command again.
   c) If you continue to get catalog compilation errors, disable environment isolation to help you isolate the error.
2. To disable environment isolation in open source Puppet:
   a) Remove the `generate types` command from any r10k hooks.
   b) Remove the `.resource_types` directory.
3. To disable environment isolation in Puppet Enterprise (PE):
   a) In `/etc/puppetlabs/puppetserver/conf.d/pe-puppet-server.conf`, remove the `pre-commit-hook-commands` setting.
   b) In Hiera, set `puppet_enterprise::master::puppetserver::pre_commit_hook_commands: []`
   c) On the command line, run `service pe-puppetserver reload`
   d) Delete the `.resource_types` directories from your staging code directory, `/etc/puppetlabs/code-staging`
   e) Deploy the environments.

## The `generate types` command

When you run the `generate types` command, it scans the entire environment for resource type implementations, excluding core Puppet resource types.

The `generate types` command accepts the following options:

- `--environment <ENV_NAME>`: The environment for which to generate metadata. If you do not specify this argument, the metadata is generated for the default environment (`production`).
- `--force`: Use this flag to overwrite all previously generated metadata.

For each resource type implementation it finds, the command generates a corresponding metadata file, named after the resource type, in the `<env-root>/.resource_types` directory. It also syncs the files in the `.resource_types` directory so that:

- Types that have been removed in modules are removed from `resource_types`.
- Types that have been added are added to `resource_types`.
- Types that have not changed (based on timestamp) are kept as is.
- Types that have changed (based on timestamp) are overwritten with freshly generated metadata.

The generated metadata files, which have a `.pp` extension, exist in the code directory. If you are using Puppet Enterprise with Code Manager and file sync, these files appear in both the staging and live code directories. The generated files are read-only. Do not delete them, modify them, or use expressions from them in manifests.

# Welcome to Puppet modules

| Helpful modules docs links | | Other useful places | |
|---|---|---|---|
| **Understanding Puppet modules** | Module fundamentals<br>Plug-ins in modules<br>Modules cheatsheet | **Modules on the Puppet Forge** | The Forge module repository<br>Puppet approved modules<br>Puppet supported modules<br>module quality scoring |
| **Managing modules** | Installing modules<br>Upgrading modules<br>Uninstalling modules<br>`puppet-module` command reference | **Modules in Code Manager** | Managing modules with the Puppetfile |
| **Writing modules** | Beginner's guide to modules<br>Module metadata<br>Publishing modules | **Puppet language reference** | Classes<br>Defined types<br>Puppet tasks and plans |
| **Documenting modules** | Writing module documentation<br>Puppet Strings<br>Puppet Strings style guide | **Developing and testing modules** | Puppet Development Kit (PDK) |
| **Publishing modules** | On the Forge | **Developing types and providers** | Puppet Resource API |
| **Contributing to modules** | Contributing to Puppet modules<br>Reviewing community pull requests | **Community resources** | Puppet Community Slack<br>puppet-users email group |

## Module fundamentals

You'll keep nearly all of your Puppet code in modules. Each module manages a specific task in your infrastructure, such as installing and configuring a piece of software. Modules serve as the basic building blocks of Puppet and are reusable and shareable.

Modules contain Puppet classes, defined types, tasks, task plans, functions, resource types and providers, and plug-ins such as custom types or facts. Modules must be installed in the Puppet modulepath. Puppet loads all content from every module in the modulepath, making this code available for use.

You can download and install modules from the Puppet Forge. The Forge contains thousands of modules written by Puppet developers and the open source community for a wide variety of use cases. You should also expect to write at least a few of your own modules to meet specific needs in your infrastructure.

If you're using Code Manager or r10k, you'll manage modules with a Puppetfile. For smaller, manually managed infrastructures or proof of concept projects, you can install and manage modules with the `puppet module` command. See the related topic about installing modules for details.

## Module structure

Modules have a specific directory structure that allows Puppet to find and load classes, defined types, facts, custom types and providers, functions, and tasks.

Each module subdirectory has a specific function. Not all directories are required, but if used, they should be in the following structure.

| | |
|---|---|
| **`data/`** | Contains data files specifying parameter defaults. |
| **`examples/`** | Contains examples showing how to declare the module's classes and defined types.<br>`init.pp`: The main class of the module.<br>`example.pp`: Provide examples for major use cases. |
| **`facts.d/`** | Contains external facts, which are an alternative to Ruby-based custom facts. These are synced to all agent nodes, so they can submit values for those facts to the Puppet master. |
| **`files/`** | Contains static files, which managed nodes can download.<br>**`service.conf`**: This file's `source => URL` is `puppet:///modules/my_module/service.conf`. Its contents can also be accessed with the `file` function, such as `content => file('my_module/service.conf')`. |
| **`functions/`** | Contains custom functions written in the Puppet language. |
| **`lib/`** | Contains plug-ins, such as custom facts and custom resource types. These are used by both the Puppet master and the Puppet agent, and they are synced to all agent nodes in the environment on each Puppet run.<br>**`facter/`**: Contains custom facts, written in Ruby.<br>**`puppet/`**: Contains custom functions, resource types, and resource providers:<br>`puppet/functions/`: Contains functions written in Ruby for the modern `Puppet::Functions` API.<br>`puppet/parser/functions/`: Contains functions written in Ruby for the legacy `Puppet::Parser::Functions` API. |

| | |
|---|---|
| | `puppet/provider/`: Contains custom resource providers written in the Puppet language.<br>`puppet/type/`: Contains custom resource types written in the Puppet language. |
| **`locales/`** | Contains files relating to module localization into languages other than English. |
| **`manifests/`** | Contains all of the manifests in the module.<br>**`init.pp`** — The `init.pp` class, if used, is the main class of the module. This class's name must match the module's name.<br>**`other_class.pp`** — Classes and defined types are named with the namespace of the module and the name of the class or defined type. For example, this class is named `my_module::other_class`.<br>**`implementation/`** — You can group related classes and defined types in subdirectories of the manifests/ directory. The name of this subdirectory is reflected in the names of the classes and types it contains. Classes and defined types are named with the namespace of the module, any subdirectories, and the name of the class or defined type.<br>`implementation/my_defined_type.pp`: This defined type is named `my_module::implementation::my`<br>`implementation/class.pp`: This defined type is named `my_module::implementation::cl` |
| **`plans/`** | Contains Puppet task plans, which are sets of tasks that can be combined with other logic. Plans are written in the Puppet language. |

| | |
|---|---|
| **`readmes/`** | The module's README localized into languages other than English. |
| **`spec/`** | Contains spec tests for any plug-ins in the lib directory. |
| **`tasks/`** | Contains Puppet tasks, which can be written in any programming language that can be read by the target node. |
| **`templates/`** | Contains templates, which the module's manifests can use to generate content or variable values.<br>**`component.erb`** — A manifest can render this template with `template('my_module/component.erb')`.<br>**`component.epp`** — A manifest can render this template with `epp('my_module/component.epp')`. |
| **`types/`** | Contains resource type aliases. |

### Module names

Module names should contain only lowercase letters, numbers, and underscores, and should begin with a lowercase letter. That is, module names should match the expression: `[a-z][a-z0-9_]*`

These restrictions are similar to those that apply to class names, with the added restriction that module names cannot contain the namespace separator (`::`), because modules cannot be nested. Certain module names are disallowed; see the list of reserved words and names.

### Manifests

Manifests, contained in the module's `manifests/` folder, each contain one class or defined type.

The `init.pp` manifest is the main class of a module and, unlike other classes or defined types, it is referred to only by the name of the module itself. For example, the class in `init.pp` in the `puppetlabs-motd` module is the `motd` class. You cannot name a class `init`.

All other classes or defined types names are composed of name segments, separated from each other by a namespace separator, `::`

- The module short name, followed by the namespace separator.
- Any `manifests/` subdirectories that the class or defined type is contained in, followed by a namespace separato.
- The manifest file name, without the extension.

For example, each module class or defined type would have the following names based on their module name and location within the `manifests/` directory:

| Module name | Filepath to class or defined type | Class or defined type name |
|---|---|---|
| `username-my_module` | `my_module/manifests/init.pp` | `my_module` |
| `username-my_module` | `my_module/manifests/other_class.pp` | `my_module::other_class` |
| `puppetlabs-apache` | `apache/manifests/security/rule_link.pp` | `apache::security::rule_link` |

| Module name | Filepath to class or defined type | Class or defined type name |
| --- | --- | --- |
| puppetlabs-apache | apache/manifests/fastcgi/server.pp | apache::fastcgi::server |

### Files in modules

You can serve files from a module's `files/` directory to agent nodes.

Download files to the agent by setting the `file` resource's `source` attribute to the `puppet:///` URL for the file. Alternately, you can access module files with the `file` function.

To download the file with a URL, use the following format for the `puppet:///` URL:

```
puppet:///<MODULE_DIRECTORY>/<MODULE_NAME>/<FILE_NAME>
```

For example, given a file located in `my_module/files/service.conf`, the URL is:

```
puppet:///modules/my_module/service.conf
```

To access files with the `file` function, pass the reference `<MODULE NAME>/<FILE NAME>` to the function, which returns the content of the requested file from the module's `files/` directory. Puppet URLs work for both `puppet agent` and `puppet apply`; in either case they retrieve the file from a module.

To learn more about the `file` function, see the function reference.

### Templates in modules

You can use ERB or EPP templates in your module to manage the content of configuration files. Templates combine code, data, and literal text to produce a string output, which can be used as the content attribute of a `file` resource or as a variable value. Templates are contained in the module's `templates/` directory.

For ERB templates, which use Ruby, use the `template` function. For EPP templates, which use the Puppet language, use the `epp` function. See the page about templates for detailed information.

The `template` and `epp` functions look up templates identified by module and template name, passed as a string in parentheses: `function('module_name/template_name.extension')`. For example:

```
template('my_module/component.erb')
```

```
epp('my_module/component.epp')
```

## Writing modules

Every Puppet user should expect to write at least some of their own modules. You must give your modules a specific directory structure and include correctly formatted metadata. Puppet Development Kit (PDK) provides tools for writing, validating, and testing modules.

PDK creates a complete module structure, class, defined type, and task templates, and configures a module testing framework. To test your modules, use PDK commands to run unit tests and to validate your module's metadata, syntax, and style. You can download and install PDK on any development machine; no Puppet installation is required. See the PDK documentation to get started.

For help getting started writing modules, see our beginner's guide to writing modules. For details on best practices and code style, see the Puppet Language style guide.

### Related information

Beginner's guide to writing modules on page 115
Create great Puppet modules by following best practices and guidelines.

Documenting modules on page 126

Document any module you write, whether your module is for internal use only or for publication on the Forge. Complete, clear documentation helps your module users understand what your module can do and how to use it.

The Puppet language style guide on page 178
This style guide promotes consistent formatting in the Puppet language, giving you a common pattern, design, and style to follow when developing modules. This consistency in code and module structure makes it easier to update and maintain the code.

## Plug-ins in modules

Puppet supports several kinds of plug-ins, which are distributed in modules. These plug-ins enable features such as custom facts and functions for managing your nodes. Modules that you download from the Forge can include these kinds of plug-ins, and you can also develop your own.

When you install a module that contains plug-ins, they are automatically enabled. At the start of every Puppet run, Puppet Server loads all the plug-ins available in the environment's modulepath. Agents download those plug-ins, so module plug-ins are available for use on the first Puppet run after you install them in an environment.

Plug-ins are available whether or not a node uses classes or defined types from a given module. In other words, even if you don't declare any classes from the `stdlib` module, nodes still use the `stdlib` custom facts. There is no way to exclude plug-ins in an environment in which they are installed.

If the agent and master are both running Puppet 5.3.4 or newer, the agent also downloads any non-English translations included in the module.

Puppet supports several kinds of plug-ins. Puppet looks for each plug-in in a different subdirectory of the module. If you are adding plug-ins to a module, be sure to place them in the correct module subdirectory. In all cases, you must name files and additional subdirectories according to the plug-in type's loading requirements.

**Important:**

Environments aren't completely isolated for certain kinds of plug-ins. If you are using custom resource types or legacy custom functions, you can encounter conflicts if your environments contain differing versions of a given plug-in.In such cases, Puppet loads the first version it encounters of the plug-in, and then continues to use that version for all environments.

To avoid plug-in conflicts for resource types, use the `puppet generate types` command as described in the environment isolation documentation. To fix issues with legacy custom functions, rewrite them with the modern API, which is not affected by this issue.

### Module plug-in types

Modules can contain different types of plug-ins, each in a specific subdirectory.

| Plug-in | Description | Used by | Module subdirectory |
|---|---|---|---|
| Custom facts | Written in Ruby, facts can provide a specified piece of information about system state. For information about writing custom facts, see the Facter custom facts documentation. | Agents only. | `lib/facter` |

| Plug-in | Description | Used by | Module subdirectory |
|---|---|---|---|
| External facts | External facts provide a way to use arbitrary executables or scripts as facts, or set facts statically with structured data. For information about external facts, see the Facter custom facts documentation. | Agents only. | `facts.d` |
| Puppet functions | Functions written in Puppet to return calculated values. For more information, see the topic about writing custom functions in Puppet. | Puppet Server only. | `functions` |
| Ruby functions | Functions written in Ruby to return calculated values. Modern Ruby functions are written for the `Puppet::Functions` API. | Puppet Server only. | `lib/puppet/functions` |
| Resource types | Written in Puppet to add new resource types to Puppet. For information about developing resource types, see custom types documentation. | Puppet Server and agents. | `lib/puppet/type` |
| Resource providers | Written in Puppet to add new resource providers to Puppet. For information about developing resource providers, see the custom providers documentation. | Puppet Server and agents. | `lib/puppet/provider` |
| Augeas lenses | Augeas provides a way to modify config files. To learn more about using Augeas with Puppet, see our Augeas tips and examples. | Agents only. | `lib/augeas/lenses` |

## Module cheat sheet

A quick reference to Puppet module terms and concepts.

For detailed explanations of Puppet module structure, terms, and concepts, see the related topics about modules.

### `manifests/`

The `manifests/` directory holds the module's Puppet code.

Each `.pp` file contains one and only one class or defined type. The filename, without the extension, is part of the full class or defined type name.

The `init.pp` manifest is unique: it contains a class or defined type that is called by the module name. For example:

`apache/manifests/init.pp`:

```
class apache {
...
}
```

Other classes and defined types are named with a `modulename::filename` convention. If a manifest is in a subdirectory of `manifests/` , the subdirectory is included as a segment of the name.

For example:

`apache/manifests/vhost.pp`:

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

`apache/manifests/config/ssl.pp`:

```
class apache::config::ssl {
...
}
```

### `files/`

You can download files in a module's `files/` directory to any node. Files in this directory are served at `puppet:///modules/modulename/filename`.

Use the `source` attribute to download file contents from the server, specifying the file with a `puppet:///` URL.

For example, to fetch `apache/files/httpd.conf` :

```
file {'/etc/apache2/httpd.conf':
  ensure => file,
  source => 'puppet:///modules/apache/httpd.conf',
```

You can also fetch files from subdirectories of `files/`. For example, to fetch `apache/files/extra/ssl`.

```
file {'/etc/apache2/httpd-ssl.conf':
  ensure => file,
  source => 'puppet:///modules/apache/extra/ssl',
}
```

### `lib/`

The `lib/` directory contains different types of Puppet plug-ins, which add features to Puppet and Facter. Each type of plug-in has its own subdirectory. For example:

The `lib/types` directory contains custom resource types:

```
apache/lib/puppet/type/apache_setting.rb
```

The `lib/puppet/functions` directory contains custom functions:

```
apache/lib/puppet/functions/apache/bool2httpd.rb
```

The `lib/facter` directory contains custom facts:

```
apache/lib/facter/apache_confdir.rb
```

#### templates/

The `templates/` directory holds ERB and EPP templates.

Templates output strings that can be used in files. To use template output for a file, set the `content` attribute to the `template` function, specifying the template in a `<modulename>/<filename>.<extension>` format.

For example, to use the `apache/templates/vhost.erb` template output as file contents:

```
file     {'/etc/apache2/sites-enabled/wordpress.conf':
  ensure => file,
  content => template('apache/vhost.erb'),
}
```

**Related information**

Module fundamentals on page 99
You'll keep nearly all of your Puppet code in modules. Each module manages a specific task in your infrastructure, such as installing and configuring a piece of software. Modules serve as the basic building blocks of Puppet and are reusable and shareable.

Plug-ins in modules on page 104
Puppet supports several kinds of plug-ins, which are distributed in modules. These plug-ins enable features such as custom facts and functions for managing your nodes. Modules that you download from the Forge can include these kinds of plug-ins, and you can also develop your own.

## Installing and managing modules from the command line

Install, upgrade, and uninstall Forge modules from the command line with the `puppet module` command.

The `puppet module` command provides an interface for managing modules from the Forge. Its interface is similar to other common package managers, such as `gem`, `apt-get`, or `yum`. You can install, upgrade, uninstall, list, and search for modules with this command.

**Restriction:** If you are using Code Manager or r10k, do not install, update, or uninstall modules with the `puppet module` command. With code management, you must install modules with a Puppetfile. Code management purges modules that were installed with the `puppet module` command. See the Puppetfile documentation for instructions.

### Setting up `puppet module` behind a proxy

To use the `puppet module` command behind a proxy, set the proxy's IP address and port by running the following two commands:

```
export http_proxy=http://<PROXY IP>:<PROXY PORT>
export https_proxy=http://<PROXY IP>:<PROXY PORT>
```

For instance, for an proxy at 192.168.0.10 on port 8080, run:

```
export http_proxy=http://192.168.0.10:8080
export https_proxy=http://192.168.0.10:8080
```

Alternatively, you can set these two proxy settings in the `puppet.conf` file, by setting `http_proxy_host` and `http_proxy_port` in the `user` section of `puppet.conf`. For more information, see the Puppetconfiguration reference.

**Important:** Set these two proxy settings only in the `[user]` section of the `puppet.conf` file. Setting them in other sections can cause problems.

## Finding Forge modules

The Forge houses thousands of modules, which you can find on the Forge website or by searching on the command line.

The easiest way to search for or browse modules is on the Forge website. Each module on the Forge has its own page with the module's quality score, community rating, and documentation. Alternatively, you can search for modules on the command line with the `puppet module search` command.

Some modules are Puppet supported or Puppet approved. Approved modules are often developed by Puppet community members and pass our specific quality and usability requirements. We recommend these modules, but they are not supported as part of a Puppet Enterprise license agreement. Puppet supported modules have been tested with PE and are fully supported. To learn more, see the Puppet approved and Puppet supported pages.

If there are no supported or approved modules that meet your needs, evaluate available modules by compatibility, documentation, last release date, number of downloads, and the module's Forge quality score.

### Searching modules from the command line

The `puppet module search` command accepts a single search term and returns a list of modules whose names, descriptions, or keywords match the search term.

For example, a search like:

```
puppet module search apache
```

returns results such as:

```
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
```

### Finding and downloading deleted modules

You can still search for and download a specific release of a module on the Forge, even if the release has been deleted.

Normally, deleted modules do not appear in Forge search results. To include deleted modules in your search on the Forge website, check **Include deleted modules** in the search filter panel.

To download a deleted release of a specific module, select the release from the **Select another release** drop-down list on the module's page. The release is marked in this menu as deleted. If you select the deleted release, a warning banner appears on the page with the reason for deletion. To download the deleted release anyway, click **Download** or install it with the `puppet module install` command.

## Installing modules from the command line

The `puppet module install` command installs a module and all of its dependencies. You can install modules from the Forge, a module repository, or a release tarball.

By default, this command installs modules into the first directory in the Puppet modulepath, which defaults to `$codedir/environments/production/modules`. For example, to install the `puppetlabs-apache` module, run:

```
puppet module install pupppetlabs-apache
```

You can customize the module version, installation directory, or environment, get debugging information, or ignore dependencies by passing options with the `puppet module install` command.

**Note:** If any installed module has an invalid version number, Puppet issues a warning:

```
Warning: module (/Users/youtheuser/.puppet/modules/module) has an invalid
version number (0.1). The version has been set to 0.0.0. If you are the
 maintainer for this module, please update the
metadata.json with a valid Semantic Version (http://semver.org).
```

Despite the warning, Puppet still downloads your module and does not permanently change the module's metadata. The version is changed only in memory during the run of the program, so that Puppet can calculate dependencies.

### Installing modules from the Forge

To install a module from the Forge, run the `puppet module install` command with the long name of the module. The long name of a module is formatted as `<username>-<modulename>`. For example, to install `puppetlabs-apache`, run:

```
puppet module install puppetlabs-apache
```

**Restriction:** On Solaris 10, when you try to install modules with the `puppet module install command`, you'll get an error like:

```
    Error: Could not connect via HTTPS to https://forgeapi.puppetlabs.com
            Unable to verify the SSL certificate
    The certificate may not be signed by a valid CA
    The CA bundle included with OpenSSL may not be valid or up to date
```

This error occurs because there is no CA-cert bundle on Solaris 10 to trust the Forge certificate. To work around this issue, download the module from the Forge website, and then install the module tarball with the `puppet module install` command, as described in the topic about installing from a release tarball.

### Installing from another module repository

You can install modules from other repositories that mimic the Forge interface. You can change the module repository for one installation, or you can change your default repository.

To change the module repository for a single module installation, specify the base URL of the repository on the command line with the `--module_repository` option. For example:

```
puppet module install --module_repository http://dev-forge.example.com
 puppetlabs-apache
```

To change the default module repository, edit the `module_repository` setting in the `puppet.conf` to the base URL of the repository you want to use. The default value for the `module_repository` is the Forge URL, `https://forgeapi.puppetlabs.com`. See the `module_repository` setting in the `puppet.conf` configuration documentation.

#### Installing from a release tarball

To install a module from a release tarball, specify the path to the tarball instead of the module name.

If you cannot connect to the Forge, or you are installing modules that have not yet been published to the Forge, use the `--ignore-dependencies` option and manually install any dependencies. For example:

```
puppet module install ~/puppetlabs-apache-0.10.0.tar.gz --ignore-
dependencies
```

#### Installing and upgrading Puppet Enterprise-only modules

Some Puppet modules are available only to PE users. Generally, you manage these modules in the same way you would manage other modules. You can use these modules with licensed PE nodes, a PE 10-node trial license, or with Bolt for a limited evaluation period. See your module's license for complete details.

### Install modules on nodes without internet

To manually install a module on a node with no internet, download the module on a connected machine, and then move a module package to the unconnected node. If the module is a PE-only module, the download machine must have a valid PE license.

Make sure you have PDK installed. You'll use PDK to build a module package that you can move to your unconnected node. For installation instructions, see the PDK install docs.

**Tip:** On machines with no internet access, you must install any module dependencies manually. Check your dependencies at the beginning of this process, so that you can move all of the necessary modules to the unconnected node at one time.

1. On a node with internet access, run `puppet module install puppetlabs-<MODULE>`
2. Change into the module's directory by running `cd <MODULE_NAME>`
3. Build a package from the installed module by running `pdk build`
4. Move the `*.tar.gz` to the machine on which you want to install the module.
5. Install the `tar.gz` package with the `puppet module install` command. For example:

   ```
   puppet module install puppetlabs-pe_module-0.1.0.tar.gz
   ```
6. Manually install the module's dependencies. Without internet access, `puppet module install` cannot install dependencies automatically.

### Upgrading modules

To upgrade a module to a newer version, use the `puppet module upgrade` command.

This command upgrades modules to the most recent released version of the module. This includes upgrading the module to the most recent major version.

Specify the module you want to upgrade with the module's full name. For example:

```
puppet module upgrade puppetlabs-apache
```

To upgrade to a specific version, specify the version you want with the `--version` option. For example, to upgrade `puppetlabs-apache` version 2.2.0 without any breaking changes, specify the 2.x release to upgrade to:

```
puppet module upgrade puppetlabs-apache --version 2.3.1
```

You can also ignore changes or dependencies when upgrading with command line options. See the `puppet module` command reference for a complete list of options.

## Uninstalling modules

Completely remove installed modules with the `puppet module uninstall` command.

This command uninstalls modules from the modulepath specified in the `puppet.conf` file. To remove a module, run the uninstall command with the full name of the module. For example:

```
puppet module uninstall puppetlabs-apache
```

By default, the command exits and returns an error if you try to uninstall a module that other modules depend on or if the module's files have been modified after it was installed. You can forcibly uninstall dependencies or changed modules with command line options.

For example, to uninstall a module that other modules depend on, run:

```
puppet module uninstall --force
```

See the `puppet module` command reference for a complete list of options.

## `puppet module` command reference

The puppet module command manages modules with several actions and options.

### `puppet module` actions

**Important:**

Solaris Note: To use `puppet module` commands on Solaris systems, you must first install `gtar`.

| Action | Description | Arguments | Example |
|---|---|---|---|
| `build` | Deprecated. Prepares a local module for release on the Forge by building a ready-to-upload archive file. Will be removed in a future release; use Puppet Development Kit instead. | A valid directory path to a module. | `puppet module build modules/apache` |
| `changes` | Compares the files on disk to the md5 checksums and returns an array of paths of modified files. | A valid directory path to a module. | `puppet module changes /etc/code/modules/stdlib` |
| `install` | Installs a module. | The full name <`username-module_name`> of the module to uninstall. | `puppet module install puppetlabs-apache` |
| `list` | Lists the modules installed in the modulepath specified in the `[main]` block in the `puppet.conf` file. | None. | `puppet module list` |
| `search` | Searches the Forge for modules matching search values. | A single search term. | `puppet module search apache` |
| `uninstall` | Uninstalls a module. | The full name <`username-module_name`> of the module to uninstall. | `puppet module uninstall puppetlabs-apache` |

| Action | Description | Arguments | Example |
| --- | --- | --- | --- |
| `upgrade` | Upgrades a module to the most recent release or to the specified version. Does not upgrade dependencies. | The full name <`username-module_name`> of the module to upgrade. | `puppet module upgrade puppetlabs-apache --version 0.0.3` |

### `puppet module install` action

Installs a module from the Forge or another specified release archive.

Usage:

```
puppet module install [--debug] [--environment] [--force | -f] [--ignore-dependencies]
  [--module_repository <REPOSITORY_URL>] [--strict-semver]
  [--target-dir <DIRECTORY/PATH> | -i <DIRECTORY/PATH>] [--version <x.x.x> |
 -v <x.x.x>]
  <full_module_name>
```

For example:

```
puppet module install --environment testing --ignore-dependencies
  --version 1.0.0-pre1 --strict-semver false puppetlabs-apache
```

| Option | Description | Value | Default |
| --- | --- | --- | --- |
| `--debug`, `-d` | Displays additional information about what the `puppet module` command is doing. | None. | If not specified, additional information is not displayed. |
| `--environment` | Installs the module into the specified environment. | An environment name. | By default, installs the module into the default environment specified in the `puppet.conf` file. |
| `--force`, `-f` | Installs the module regardless of dependency tree, checksum changes, or whether the module is already installed. By default, installs the module in the default modulepath, even if the module is already installed in another directory. Does not install dependencies. | None. | If not specified, `puppet module install` exits and returns information if it encounters installation errors or conflicts. |
| `--ignore-dependencies` | Does not install any modules required by this module. | None. | If not specified, the `puppet module install` action installs the module and its dependencies. |

| Option | Description | Value | Default |
|---|---|---|---|
| `--module_repository` | Specifies a module repository. | A valid URL for a module repository. | If not specified, installs modules from the module repository specified in from the `puppet.conf` file. By default, this is the URL for the Forge. |
| `--strict-semver` | Whether to exclude pre-release versions. A value of false allows installation of pre-release versions. | `true`, `false` | Defaults to `true`, excluding pre-release versions. |
| `--target-dir`, `-i` | Specifies a directory to install modules. | A valid directory path. | By default, installs modules into `$codedir/environments/production/modules` |
| `--version`, `-v` | Specifies the module version to install. | A semantic version number, such as 1.2.1 or a string specifying a requirement, such as ">=1.0.3". | If not specified, installs the most recent version available on the Forge. |

#### `puppet module list` action

Lists the Puppet modules installed in the modulepath specified in the `puppet.conf` file's `[main]` block. Use the `--modulepath` option to change which directories are scanned.

Usage:

```
puppet module list [--tree] [--strict-semver]
```

For example:

```
puppet module list --tree --modulepath etc/testing/modules
```

| Option | Description | Value | Default |
|---|---|---|---|
| `--modulepath` | Specifies another modulepath to scan for modules. | A valid directory path. | By default, scans the default modulepath from the `[main]` block in the `puppet.conf` file. |
| `--strict-semver` | Whether to exclude pre-release versions. A value of false allows uninstallation of pre-release versions. | `true`, `false` | Defaults to `true`, excluding pre-release versions. |
| `--tree` | Displays the module list as a tree showing dependencies. | None. | By default, `puppet module list` lists installed modules but does not show dependency relationships. |

#### `puppet module uninstall` action

Uninstalls a module from the default modulepath.

Usage:

```
puppet module uninstall [--force | -f] [--ignore-changes | -c] [--strict-
semver] [--version=]  <full_module_name>
```

For example:

```
puppet module uninstall --ignore-changes --version 0.0.2 puppetlabs-apache
```

| Option | Description | Value | Default |
|---|---|---|---|
| `--force`, `-f` | Uninstalls the module regardless of dependency tree or checksum changes. | None. | By default, `puppet module uninstall` exits and returns an error if it encounters changes, namespace errors, or dependencies. |
| `--ignore-changes` | Does not use the checksum and uninstalls regardless of modified files. | None. | By default, if the `puppet module uninstall` action finds modified files in the module, it exits and returns an error. |
| `--strict-semver` | Whether to exclude pre-release versions. A value of false allows uninstallation of pre-release versions. | `true`, `false` | Defaults to `true`, excluding pre-release versions. |
| `--version`, `-v` | Specifies the module version to uninstall. | A semantic version number, such as 1.2.1 or a string specifying a requirement, such as ">=1.0.3".. | By default, `puppet module uninstall` uninstalls the version installed in the modulepath. |

#### `puppet module upgrade` action

This command upgrades modules to the most recent released version of the module. This includes upgrades to the most recent major version.

Usage:

```
puppet module upgrade [--force | -f] [--ignore-changes | -c] [--ignore-
dependencies]
  [--strict-semver] [--version=] <full_module_name>
```

For example:

```
puppet module upgrade --force --version 2.1.2 puppetlabs-apache
```

| Option | Description | Value | Default |
|---|---|---|---|
| `--force`, `-f` | Upgrades the module regardless of dependency tree or checksum changes. | None. | By default, `puppet module upgrade` exits and returns an error if it encounters changes, namespace errors, or dependencies. |

| Option | Description | Value | Default |
|---|---|---|---|
| `--ignore-changes` | Does not use the checksum and upgrades regardless of modified files. | None. | By default, if the `puppet module upgrade` action finds modified files in the module, it exits and returns an error. |
| `--ignore-dependencies` | Does not attempt to install any missing modules required by this module. | None. | If not specified, the `puppet module upgrade` action installs missing module dependencies. |
| `--strict-semver` | Whether version ranges should exclude pre-release versions. | `true`, `false` | Defaults to `true`, excluding pre-release versions. |
| `--version`, `-v` | Specifies the module version to uninstall. | A semantic version number, such as 1.2.1 or a string specifying a requirement, such as ">=1.0.3". | By default, `puppet module uninstall` uninstalls the version installed in the modulepath. |

### PE-only module troubleshooting

If you get an error when installing a PE-only module, check for common issues.

When installing or upgrading a PE-only module, you might get the following error:

```
Error: Request to Puppet Forge failed.
  The server being queried was https://forgeapi.puppetlabs.com/v3/releases?
module=puppetlabs-f5&module_groups=base+pe_only
  The HTTP response we received was '403 Forbidden'
  The message we received said 'You must have a valid Puppet Enterprise
 license on this
  node in order to download puppetlabs-f5. If you have a Puppet Enterprise
 license,
  please see https://docs.puppetlabs.com/pe/latest/
modules_installing.html#puppet-enterprise-modules
  for more information.'
```

If you aren't a PE user, you won't be able to use this module unless you purchase a PE license. If you are a PE user, check the following:

1. Are you logged in as the root user? If not, log in as root and try again.
2. Does the node you're on have a valid PE license? If not, switch to a node that has a valid license on it.
3. Are you running a version of PE that supports this module? If not, you might need to upgrade.
4. Does the node you are installing on have access to the internet? If not, switch to a node that has access to the internet.

## Beginner's guide to writing modules

Create great Puppet modules by following best practices and guidelines.

This guide is intended to provide an approachable introduction to module best practices. Before you begin, we recommend that you are familiar enough with Puppet that you have a basic understanding of the language, you know what constitutes a class, and you understand the basic module structure.

### Defining your module

Before you begin writing your module, define what it will do. Defining the range of your module's work helps you create concise modules that are easy to work with. A good module has only one area of responsibility. For example, the module addresses installing MySQL, but it doesn't install other programs or services that require MySQL.

Ideally, a module manages a single piece of software from installation through setup, configuration, and service management. When you plan your module, consider what task your module will accomplish and what functions it requires in your Puppet environment. Many users have 200 or more modules in an environment, so simple is better. For more complex needs, create multiple modules. Having many small, focused modules promotes code reuse and turns modules into building blocks.

For example, the `puppetlabs-puppetdb` module deals solely with the the setup, configuration, and management of PuppetDB. However, PuppetDB stores its data in a PostgreSQL database. Instead of trying to manage PostgreSQL with the `puppetdb` module, we included the `puppetlabs-postgresql` module as a dependency. This way, the `puppetdb` module can use the `postgresql` module's classes and resources to build out the right configuration.

### Class design

A good module is made up of small, self-contained classes that each do only one thing. Classes within a module are similar to functions in programming, using parameters to perform related steps that create a coherent whole.

In general, files must have the same named as the class or definition that it contains, and classes must be named after their function. The one exception to this rule is the main class of a module, which is defined in the `init.pp` file, but is called by the same name as the module. Generally, a module includes:

- The `<MODULE>` class: The main class of the module shares the name of the module and is defined in the `init.pp` file.
- The `install` class: Contains all of the resources related to installing the software that the module manages.
- The `config` class: Contains resources related to configuring the installed software.
- The `service` class: Contains service resources, as well as anything else related to the running state of the software.

For more information and an example of this structure and the code contained in classes, see the topic about module classes.

### Parameters

Parameters form the public API of your module. They are the most important interface you expose, so be sure to balance to the number and variety of parameters so that users can customize their interactions with the module.

Name your parameters in a consistent `thing_property` pattern, such as `package_ensure`. Consistency in names helps users understand your parameters and aids in troubleshooting and collaborative development. If you have a parameter that manages the entire installation of a package, you can use the `package_manage` convention. The `package_manage` pattern allows you to wrap all of the resources in an `if $package_manage {}` test, as shown in this `ntp` example:

```
class ntp::install {

  if $ntp::package_manage {
    package { $ntp::package_name:
      ensure => $ntp::package_ensure,
    }
  }
}
```

To make sure users can customize your module as needed, add parameters. Do not hardcode data in your module, because this makes it inflexible and harder to use in even slightly different circumstances. For the same reason, avoid adding parameters that allow users to override templates. When you allow template overrides, users can override your template with a custom template containing additional hardcoded parameters. Instead, it's better to add flexible, user configurable parameters as needed.

For an example of a module that offers many parameters to increase flexibility, see the puppetlabs-apache module.

### Ordering

Base all order-related dependencies (such as `require` and `before`) on classes rather than resources. Class-based ordering allows you to isolate the implementation details of each class. For example, rather than specifiying `require` for several packages, you can use one class dependency. This allows you to make adjustments to the `module::install` class only, instead of adjusting multiple class manifests:

```
file { 'configuration':
      ensure  => present,
      require => Class['module::install'],
    }
```

### Containment

Ensure that your main classes explicitly contain any subordinate classes they declare. Classes do not automatically contain the classes they declare, because classes can be declared in several places via `include` and similar functions. If your classes contain the subordinate classes, it makes it easier for other modules to form ordering relationships with your module.

To contain classes, use the `contain` function. For example, the `puppetlabs-ntp` module uses containment in the main `ntp` class:

```
contain ntp::install
contain ntp::config
contain ntp::service

Class['ntp::install']
-> Class['ntp::config']
~> Class['ntp::service']
```

For more information about containment, see the containment documentation.

### Dependencies

If your module's functionality depends on another module, list these dependencies in the module and include them directly in the module's main class with an `include` statement. This ensures that the dependency is included in the catalog. List the dependency to the module's `metadata.json` file and the `.fixtures.yml` file used for RSpec unit testing.

### Testing modules

Test your module to make sure that it works in a variety of conditions and that its options and parameters work together. PDK includes tools for validating and running unit tests on your module, including RSpec, RSpec Puppet, and Puppet Spec Helper.

Write unit tests to verify that your module works as intended in a variety of circumstances. For example, to ensure that the module works in different operating systems, write tests that call the `osfamily` fact to verify that the package and service exist in the catalog for each operating system your module supports.

To learn more about how to write unit tests, see the RSpec testing tutorial. For more information on testing tools, see the tools list below.

| | |
|---|---|
| **`rspec-puppet`** | Extends the RSpec testing framework to understand and work with Puppet catalogs, the artifact it specializes in testing. This allows you to write tests that verify that your module works as intended. This tool is included in PDK. |

| | |
|---|---|
| | For example, you can call facts, such as `osfamily`, with RSpec, iterating over a list of operating systems to make sure that the package and service exist in the catalog for every operating system your module supports.<br><br>To learn more about `rspec-puppet` use and unit testing, see the rspec-puppet page. |
| **`puppetlabs_spec_helper`** | Automates some of the tasks required to test modules. This is especially useful in conjunction with `rspec-puppet`, because `puppetlabs_spec_helper` provides default Rake tasks that allow you to standardize testing across modules. It also provides some code to connect `rspec-puppet` with modules. This tool is included in PDK.<br><br>To learn more, see the puppetlabs_spec_helper project. |
| **`beaker-rspec`** | An acceptance and integration testing framework. It provisions one or more virtual machines on various hypervisors (such as Vagrant) and then checks the result of applying your module in a realistic environment. To learn more, see the beaker-spec project. |
| **`serverspec`** | Provides additional testing constructs (such as `be_running` and `be_installed`) for `beaker-rspec`. Serverspec allows you to test against different distributions by executing test commands locally. To learn more, see the Serverspec site. |

### Documenting your module

Document your module's use cases, usage examples, and parameter details with `README.md` and `REFERENCE.md` files. In the README, explain why and how users would use your module, and provide usage examples. Use Puppet Strings to create the REFERENCE, which is a detailed list of information about your module's classes, defined types, functions, tasks, task plans, and resource types and providers. For more about writing your README and creating the REFERENCE, see our module documentation guide and the Strings documentation.

### Versioning your module

Whenever you make changes to your module, update the version number. Version your module semantically to help users understand the level of changes in your updated module. To learn more about the specific rules of semantic versioning, see the semantic versioning specification.

After you've decided on the new version number, adjust the version number in the `metadata.json` file. This allows you to create a list of dependencies in the `metadata.json` file of your modules with specific versions of dependent modules, which ensures your module isn't used with an old dependency that won't work. Versioning also enables workflow management by allowing you to easily use different versions of modules in different environments.

### Releasing your module

Publish your modules on the Forge to share your modules with other Puppet users. Sharing modules allows other users to not only download and use your module to solve their infrastructure problems, but also to contribute their own improvements to your modules. Sharing modules fosters community among Puppet users, and helps improve the quality of modules available to everyone. To learn how to publish your modules to the Forge, see the module publishing documentation.

## Module classes

A typical module contains a main module class, as well as classes for managing installation, configuration, and the running state of the managed software. The `puppetlabs-ntp` module provides examples of the classes in such a module structure.

### `module`

The main class of any module shares the name of the module, but the file itself if named `init.pp`. This class is the module's main interface point with Puppet, and if possible, you should make it the only parameterized class in your module. Limiting the parameterized classes to the main class only means that you only have to include a single class to control usage of the entire module. This class should provide sensible defaults so that a user can get going by just declaring the main class with `include module`.

For instance, the main `ntp` class in the `puppetlabs-ntp` module is the only parameterized class in the module:

```
class ntp (
  Boolean $broadcastclient,
  Stdlib::Absolutepath $config,
  Optional[Stdlib::Absolutepath] $config_dir,
  String $config_file_mode,
  Optional[String] $config_epp,
  Optional[String] $config_template,
  Boolean $disable_auth,
  Boolean $disable_dhclient,
  Boolean $disable_kernel,
  Boolean $disable_monitor,
  Optional[Array[String]] $fudge,
  Stdlib::Absolutepath $driftfile,
 ...
```

### `module::install`

The install class must be located in the `install.pp` file. It should contain all of the resources related to getting the software that the module manages onto the node. The install class must be named `module::install`. In the `puppetlabs-ntp` module, this class is private, which means users do not interact with the class directly.

```
class ntp::install {

  if $ntp::package_manage {
    package { $ntp::package_name:
      ensure => $ntp::package_ensure,
    }
  }
}
```

### `module::config`

The resources related to configuring the installed software should be placed in a config class. The config class must be named `module::config` and must be located in the `config.pp` file. In the `puppetlabs-ntp` module, this class is private, which means users do not interact with the class directly.

```
class ntp::config {

  # The servers-netconfig file overrides NTP config on SLES 12, interfering
 with our configuration.
  if $facts['operatingsystem'] == 'SLES' and
 $facts['operatingsystemmajrelease'] == '12' {
    file { '/var/run/ntp/servers-netconfig':
      ensure => 'absent'
    }
```

```
  }

  if $ntp::keys_enable {
    case $ntp::config_dir {
      '/', '/etc', undef: {}
      default: {
        file { $ntp::config_dir:
          ensure  => directory,
          owner   => 0,
          group   => 0,
          mode    => '0775',
          recurse => false,
        }
      }
    }

    file { $ntp::keys_file:
      ensure  => file,
      owner   => 0,
      group   => 0,
      mode    => '0644',
      content => epp('ntp/keys.epp'),
    }
  }
...
```

**`module::service`**

The remaining service resources, and anything else related to the running state of the software, should be contained in the service class. The service class must be named `module::service` and must be located in the `service.pp` file. In the `puppetlabs-ntp` module, this class is private, which means users do not interact with the class directly.

```
class ntp::service {

  if ! ($ntp::service_ensure in [ 'running', 'stopped' ]) {
    fail('service_ensure parameter must be running or stopped')
  }

  if $ntp::service_manage == true {
    service { 'ntp':
      ensure     => $ntp::service_ensure,
      enable     => $ntp::service_enable,
      name       => $ntp::service_name,
      provider   => $ntp::service_provider,
      hasstatus  => true,
      hasrestart => true,
    }
  }
}
```

## Module metadata

Your module must contain certain metadata in a `metadata.json` file, which tracks important information about the module.

The `metadata.json` file is located in the module's main directory, outside any subdirectories. If you created your module with Puppet Development Kit (PDK), the `metadata.json` file is already created and contains the information you provided during the module creation interview. If you skipped the interview, the module metadata is populated with PDK default values. You can manually edit the values in the `metadata.json` file as needed.

The Forge requires modules to contain the `metadata.json` file. The Forge uses the metadata to create the module's information page and to provide important information to users installing the module. The `metadata.json` file uses standard JSON syntax and contains a single JSON object, mapping keys to values.

`metadata.json` example

```
{
  "name": "puppetlabs-ntp",
  "version": "6.1.0",
  "author": "Puppet Inc",
  "summary": "Installs, configures, and manages the NTP service.",
  "license": "Apache-2.0",
  "source": "https://github.com/puppetlabs/puppetlabs-ntp",
  "project_page": "https://github.com/puppetlabs/puppetlabs-ntp",
  "issues_url": "https://tickets.puppetlabs.com/browse/MODULES",
  "dependencies": [
    { "name":"puppetlabs/stdlib","version_requirement":">= 4.13.1 < 5.0.0" }
  ],
  "data_provider": "hiera",
  "operatingsystem_support": [
    {
      "operatingsystem": "RedHat",
      "operatingsystemrelease": [
        "5",
        "6",
        "7"
      ]
    },
    {
      "operatingsystem": "CentOS",
      "operatingsystemrelease": [
        "5",
        "6",
        "7"
      ]
    }
  ],
  "requirements": [
    {
      "name": "puppet",
      "version_requirement": ">= 4.5.0 < 5.0.0"
    }
  ],
  "description": "NTP Module for Debian, Ubuntu, CentOS, RHEL, OEL, Fedora,
 FreeBSD, ArchLinux, Amazon Linux and Gentoo."
}
```

### Specifying dependencies

If your module depends on functionality from another module, specify this in the `"dependencies"` key of the `metadata.json` file. The `"dependencies"` key accepts an array of hashes. This key is required, but if your module has no dependencies, you can pass an empty array.

Dependencies are not added to the metadata during module creation, so you must edit your `metadata.json` file to include dependency information. For information about how to format dependency versions, see the related topic about version specifiers in module metadata.

The hash for each dependency must contain the `"name"` and `"version_requirement"` keys. For example:

```
"dependencies": [
  { "name": "puppetlabs/stdlib", "version_requirement": ">= 3.2.0 <
 5.0.0" },
```

```
  { "name": "puppetlabs/firewall", "version_requirement": ">= 0.0.4" },
  { "name": "puppetlabs/apt", "version_requirement": ">= 1.1.0 < 2.0.0" },
  { "name": "puppetlabs/concat", "version_requirement": ">= 1.0.0 < 2.0.0" }
]
```

If you install modules with the `puppet module install` command, Puppet installs any missing dependencies. If you install modules with Code Manager and the Puppetfile, dependencies are not automatically installed, so you must specify them in your Puppetfile.

### Specifying Puppet version requirements

The `requirements` key specifies external requirements for the module, particularly the Puppet version required. Although you can express any requirement here, the Forge module pages and search function support only the `"puppet"` value, which specifies the Puppet version.

The `"requirements"` key accepts an array of hashes with the following keys:

- `"name"`: The name of the requirement.
- `"version_requirement"`: A semantic version range, including lower and upper version bounds.

```
"requirements": [
  {"name": "puppet”, “version_requirement”: ">= 4.5.0 < 5.0.0"}
]
```

**Important:** The Forge requires both lower and upper bounds for the Puppet version requirement. If you upload a module that does not specify an upper bound, the Forge adds an upper bound of the next major version. For example, if you upload a module that specifies a lower bound of 4.5.0 and no upper bound, the Forge applies an upper bound of `< 5.0.0`.

For Puppet Enterprise versions, specify the core Puppet version included in that version of PE. For example, PE 2017.1 contained Puppet 4.9. Do not express requirements for Puppet versions earlier than 3.0, because those versions do not follow semantic versioning. For information about formatting version requirements, see the related topic about version specifiers in module metadata.

### Specifying operating system compatibility

Specify the operating system your module is compatible with in the `operatingsystem_support` key. This key accepts an array of hashes, where each hash contains `operatingsystem` and `operatingsystemrelease` keys. The Forge uses these keys for search filtering and to display versions on module pages.

- The `operatingsystem` key accepts a string. The Forge uses this value for search filters.
- The `operatingsystemrelease` accepts an array of strings. The Forge displays these versions on module pages, and you can format them in whatever way makes sense for the operating system in question.

For example:

```
"operatingsystem_support": [
  {
  "operatingsystem":"RedHat",
  "operatingsystemrelease":[ "5.0", "6.0" ]
  },
  {
  "operatingsystem": "Ubuntu",
  "operatingsystemrelease": [
    "12.04",
    "10.04"
    ]
  }
]
```

### Specifying versions

Your module metadata specifies your own module's version as well as the versions for your module's dependencies and requirements. Version your module semantically; for details about semantic versioning (also known as SemVer), see the Semantic Versioning specification. This helps others know what to expect from your module when you make changes.

When you specify versions for a module dependencies or requirements, you can specify multiple versions.

If your module is compatible with only one major or minor version, use the semantic major and minor version shorthand, such as 1.x or 1.2.1. If your module is compatible with multiple major versions, you can set a supported version range.

For example, 1.x indicates that your module is compatible with any minor update of version 1, but is not compatible with version 2 or larger. Specifying a version range such as >= 1.0.0 < 3.0.0 indicates the the module is compatible with any version that greater than or equal to 1.0.0 and less than 3.0.0.

Always set an upper version boundary in your version range. If your module is compatible with the most recent released versions of a dependencies, set the upper bound to exclude the next, unreleased major version. Without this upper bound, users might run into compatibility issues across major version boundaries, where incompatible changes occur.

For example, to accept minor updates to a dependency but avoid breaking changes, specify a major version. This example accepts any minor version of `puppetlabs-stdlib` version 4:

```
"dependencies": [
  { "name": "puppetlabs/stdlib", "version_requirement": "4.x" },
]
```

In the example below, the current version of `puppetlabs-stdlib` is 4.8.0, and version 5.0.0 is not yet released. Because 5.0.0 might have breaking changes, the upper bound of the version dependency is set to that major version.

```
"dependencies": [
  { "name": "puppetlabs/stdlib", "version_requirement": ">= 3.2.0 < 5.0.0" }
]
```

The version specifiers allowed in module dependencies are:

| Format | Description |
|---|---|
| 1.2.3 | A specific version. |
| 1.x | A semantic major version. This example includes 1.0.1 but not 2.0.1. |
| 1.2.x | A semantic major and minor version. This example includes 1.2.3 but not 1.3.0. |
| > 1.2.3 | Greater than the specified version. |
| < 1.2.3 | Less than the specified version. |
| >= 1.2.3 | Greater than or equal to the specified version. |
| <= 1.2.3 | Less than or equal to the specified version. |
| >= 1.0.0 < 2.0.0 | Range of versions; both conditions must be satisfied. This example includes version 1.0.1 but not version 2.0.1. |

**Note:** You cannot mix semantic versioning shorthand (such as .x) with syntax for greater than or less than versioning. For example, you could not specify `">= 3.2.x < 4.x"`

### Adding tags

Optionally, you can add tags to your metadata to help users find your module in Forge searches. Generally, include four to six tags for any given module.

Pass tags as an array, like `["msyql", "database", "monitoring"]`.Tags cannot contain whitespace. Certain tags are prohibited, such as profanity or tags resembling the `$::operatingsystem` fact (such as `"redhat"`, `"rhel"`, `"debian"`, `"windows"`, or `"osx"`). Use of prohibited tags lowers your module's quality score on the Forge.

## Available `metadata.json` keys

Required and optional `metadata.json` keys specify metadata for your module.

| Key | Required? | Value | Example |
|---|---|---|---|
| `"name"` | Required. | The full name of your module, including your Forge username, in the format `username-module`. | `"puppetlabs-stdlib"` |
| `"version"` | Required. | The current version of your module. This should follow semantic versioning. For details, see the Semantic Versioning specification. | `"1.2.1"` |
| `"author"` | Required. | The person who gets credit for creating the module. If absent, this key defaults to the username portion of the name key. | `"puppetlabs"` |
| `"license"` | Required. | The license under which your module is made available. License metadata should match an identifier provided by SPDX. For a complete list, see the SPDX license list. | `"Apache-2.0"` |
| `"summary"` | Required. | A one-line description of your module. | `"Standard library of resources for Puppet modules."` |

| Key | Required? | Value | Example |
|---|---|---|---|
| "source" | Required. | The source repository for your module. | "https://github.com/puppetlabs/puppetlabs-stdlib" |
| "dependencies" | Required. | An array of other modules that your module depends on to function. If the module has no dependencies, pass an empty array. See the related topic about specifying dependencies for more details. | `"dependencies": [ { "name": "puppetlabs/stdlib", "version_requirement": ">= 4.13.1 < 6.0.0" } ],` |
| "requirements" | Optional. | A list of external requirements for your module, given as an array of hashes. | `"requirements": [ { "name": "puppet", "version_requirement": ">= 4.7.0 < 6.0.0" } ],` |
| "project_page" | Optional. | A link to your module's website, to be included on the module's Forge page. | "https://github.com/puppetlabs/puppetlabs-stdlib" |
| "issues_url" | Optional. | A link to your module's issue tracker. | "https://tickets.puppetlabs.com/browse/MODULES" |
| "operatingsystem_support" | Optional. | An array of hashes listing the operating systems that your module is compatible with. See the topic about specifying operating compatibility for details. | `{ "operatingsystem": "RedHat", "operatingsystemrelease": [ "5", "6", "7" ] }` |
| "tags" | Optional. | An array of four to six key words to help people find your module. | ["msyql", "database", "monitoring", "reporting"] |

# Documenting modules

Document any module you write, whether your module is for internal use only or for publication on the Forge. Complete, clear documentation helps your module users understand what your module can do and how to use it.

Write your module usage documentation in Markdown, in a README based on our module README template. Use Puppet Strings to generate reference information for your module's classes, defined types, functions, tasks, task plans, and resource types and providers.

Module documentation should be clear, consistent, and readable. It should be easy to read both on the web and in terminal. Whether you are writing your README or code comments for Puppet Strings docs generation, following some basic formatting guidelines and best writing practices can help make your module documentation great.

### Documentation best practices

If you want your documentation to really shine, a few best practices can help make your documentation clear and readable.

- Use the second person; that is, write directly to the person reading your document. For example, “If you’re installing the cat module on Windows....”
- Use the imperative; that is, directly tell the user what they should do. For example, "Secure your dog door before installing the cat module."
- Use the active voice whenever possible. For example, "Install the cat and bird modules on separate instances" rather than "The cat and bird modules should be installed on separate instances."
- Use the present tense, almost always. Events that regularly occur should be present tense: "This parameter sets your cat to 'purebred'. The purebred cat alerts you for breakfast at 6 a.m." Use future tense only when you are specifically referring to something that takes place at a time in the future, such as "The `tail` parameter is deprecated and will be removed in a future version. Use `manx` instead."
- Avoid subjective words. For example, don't write "It's quick and easy to teach an old cat new tricks." Subjective words like "quick" and "easy" can frustrate and even alienate a reader who finds teaching a cat difficult or time-consuming.
- Lists, whether ordered or unordered, make things clearer for the reader. When steps should happen in a sequence, use an ordered list (1, 2, 3…). If order doesn’t matter, like in a list of options or requirements, use an unordered (bulleted) list.

**Related information**

Documenting modules with Puppet Strings on page 131
Produce complete, user-friendly module documentation by using Puppet Strings. Strings uses tags and code comments, along with the source code, to generate documentation for a module's classes, defined types, functions, tasks, plans, and resource types and providers.

Puppet Strings style guide on page 137
To document your module with Puppet Strings, add descriptive tags and comments to your module code. Write consistent, clear code comments, and include at least basic information about each element of your module (such as classes or defined types).

## Writing the module README

In your README, include basic module information and extended usage examples for the most common use cases.

Your README should tell users what your module does and how they can use it. Include reference information as a separate `REFERENCE.md` file in the module's root directory.

**Important:** The Reference section of the README is deprecated; module READMEs should no longer contain extensive reference information. Puppet Strings generates a `REFERENCE.md` file containing all the reference information for your module, including a complete list of your module's classes, defined types, functions, resource types and providers, Puppet tasks and plans, along with parameters for each. See the topic about creating reference documentation for details.

Write your README in Markdown and use the `.md` or `.markdown` extension for the file. If you used Puppet Development Kit (PDK), you already have a copy of the README template in `.md` format in your module. For more information about Markdown usage, see the Commonmark reference.

The README should contain the following sections:

| | |
|---|---|
| **Description** | What the module does and why it is useful. |
| **Setup** | Prerequisites for module use and getting started information. |
| **Usage** | Instructions and examples for common use cases or advanced configuration options. |
| **Reference** | If the module contains facts or type aliases, include them in a short supplementary reference section. All other reference information, such as classes and their parameters, are in the `REFERENCE.md` file generated by Strings. |
| **Limitations** | OS compatibility and known issues. |
| **Development** | Guide for contributing to the module. |

### Table of contents

The table of contents helps your users find their way around your module README.

Start with the module name as a Level 1 heading at the top of the module, followed by "Table of Contents" as a Level 4 heading. Under the table of contents heading, include a numbered list of top-level sections, with any necessary subsections in a bulleted list below the section heading. Link each section to its corresponding heading in the README.

```
# modulename

#### Table of Contents

1. [Module Description - What the module does and why it is useful](#module-
description)
1. [Setup - The basics of getting started with [modulename]](#setup)
   * [What [modulename] affects](#what-[modulename]-affects)
   * [Setup requirements](#setup-requirements)
   * [Beginning with [modulename]](#beginning-with-[modulename])
1. [Usage - Configuration options and additional functionality](#usage)
1. [Limitations - OS compatibility, etc.](#limitations)
1. [Development - Guide for contributing to the module](#development)
```

### Module description

In your module description, briefly tell users why they might want to use your module. Explain what your module does and what kind of problems users can solve with it.

This should be a fairly short description helps the user decide if your module is what they want. What are the most common use cases for your module? Does your module just install software? Does it install and configure it? Give your user information about what to expect from the module.

```
## Module description

The `cat` module installs, configures, and maintains your cat in both
 apartment and residential house settings.
```

```
The cat module automates the installation of a cat to your apartment or
 house, and then provides options for configuring the cat to fit your
 environment's needs. After it's installed and configured, the cat module
 automates maintenance of your cat through a series of resource types and
 providers.
```

### Setup section

In the setup section, detail how your user can successfully get your module functioning. Include requirements, steps to get started, and any other information users might need to know before they start using your module.

Module installation instructions are covered both on the module's Forge page and in the Puppet docs, so don't reiterate them here. In this section, include the following subsections, as applicable:

**What <modulename> affects**

Include this section only if:

- The module alters, overwrites, or otherwise touches files, packages, services, or operations other than the named software; OR
- The module's general performance can overwrite, purge, or otherwise remove entries, files, or directories in a user's environment. For example:

```
## Setup

### What cat affects

* Your dog door might be
 overwritten if not secured before
 installation.
```

**Setup requirements**

Include this section only if the module requires additional software or some tweak to a user's environment. For instance, the `puppetlabs-firewall` module uses Ruby-based providers which required `pluginsync` to be enabled.

**Beginning with <modulename>**

Always include this section to explain the minimum steps required to get the module up and running in a user's environment. You can use basic proof of concept use cases here; it doesn't have to be something you would run in production. For simple modules, "Declare the main `::cat` class" is enough.

### Usage section

Include examples for common use cases in the usage section. Provide usage information and code examples to show your users how to use your module to solve problems.

If there are many use cases for your module, include three to five examples of the most important or common tasks a user can accomplish. The usage section is a good place to include more complex examples that involve different types, classes, and functions working together. For example, the usage section for the `puppetlabs-apache` module includes an example for setting up a virtual host with SSL, which involves several classes.

```
## Usage

You can manage all interaction with your cat through the main `cat`
 class. With the default options, the module installs a basic cat with no
 optimizations.
```

````
### I just want cat, what's the minimum I need?

```
include '::cat'
```

### I want to configure my lasers

Use the following to configure your lasers for a random-pattern, 20-minute
 playtime at 3 a.m. local time.

```
    class { 'cat':
      laser => {
        pattern    => 'random',
        duration   => '20',
        start_time => '0300',
      }
    }
```
````

#### Limitations section

In the limitations section, list any incompatibilities, known issues, or other warnings.

```
## Limitations

This module cannot be used with the smallchild module.
```

#### Development section

In the development section, tell other users the ground rules for contributing to your project and how they should submit their work.

## Creating reference documentation

List reference information --- a complete list of classes, defined types, functions, resource types and providers, tasks, and plans --- in a separate `REFERENCE.md` file in the root directory of your module.

Use Puppet Strings to generate this documentation based on your comments and module code. If you aren't yet using Strings to generate documentation, you can manually create a `REFERENCE.md` file.

**Tip:** Previously, we recommended that module authors include reference information in the README itself. However, the reference section often became quite long and difficult to maintain. Moving reference information to a separate file keeps the README more readable, and using Strings to generate this file makes it easier to maintain.

The Forge displays information from a module's `REFERENCE.md` file in a reference tab on the module's detail page, so the information remains easily accessible to users. To create a `REFERENCE.md` file for your module, add Strings comments to the code for each of your classes, defined types, functions, task plans, and resource types and providers, and then run Strings to generate documentation in Markdown. You can create a `REFERENCE.md` file manually, but remember that if you then generate a `REFERENCE.md` with Strings, it overwrites any existing `REFERENCE.md` file.

For details on adding comments to your code, see the Strings style guide. For instructions on how to install and use Strings, see the topics about Puppet Strings.

#### Manually writing reference documentation

If you aren't using Strings yet to generate your reference documentation, you can manually create a `REFERENCE.md` file listing each of your classes, defined types, resource types and providers, functions, and facts, along with any parameters.

To manually document reference information, start your reference document with a small table of contents that first lists the classes, defined types, and resource types of your module. If your module contains both public and private classes or defined types, list the public and the private separately. Include a brief description of what these items do in your module.

```
## Reference

### Classes

#### Public classes

*[`pet::cat`](#petcat): Installs and configures a cat in your environment.

#### Private classes

*[`pet::cat::install`]: Handles the cat packages.
*[`pet::cat::configure`]: Handles the configuration file.
```

After this table of contents, list the parameters, providers, or features for each element (class, defined type, function, and so on) of your module. Be sure to include valid or acceptable values and any defaults that apply. Each element in this list should include:

- The data type, if applicable.
- A description of what the element does.
- Valid values, if the data type doesn't make it obvious.
- Default value, if any.

```
### `pet::cat`

#### Parameters

##### `purr`

Data type: Boolean.

Enables purring in your cat.

Default: `true`.

##### `meow`

Enables vocalization in your cat. Valid options: 'string'.

Default: 'medium-loud'.

#### `laser`

Specifies the type, duration, and timing of your cat's laser show.

Default: `undef`.

Valid options: A hash with the following keys:

* `pattern` - accepts 'random', 'line', or a string mapped to a custom
 laser_program, defaults to 'random'.
* `duration` - accepts an integer in seconds, defaults to '5'.
* `frequency` - accepts an integer, defaults to 1.
* `start_time` - accepts an integer specifying the 24-hr formatted start
 time for the program.
```

# Documenting modules with Puppet Strings

Produce complete, user-friendly module documentation by using Puppet Strings. Strings uses tags and code comments, along with the source code, to generate documentation for a module's classes, defined types, functions, tasks, plans, and resource types and providers.

If you are a module author, add descriptive tags and comments with the code for each element (class, defined type, function, or plan) in your module. Strings extracts information from the module's Puppet and Ruby code, such as data types and attribute defaults. Whenever you update code, update your documentation comments at the same time. Both module users and authors can generate module documentation with Strings. Even if the module contains no code comments, Strings generates minimal documentation based on the information it can extract from the code.

Strings outputs documentation in HTML, JSON, or Markdown formats.

- HTML output, which you can read in any web browser, includes the module README and reference documentation for all classes, defined types, functions, tasks, task plans, and resource types.
- JSON output includes the reference documentation only, and writes it to either `STDOUT` or to a file.
- Markdown output includes the reference documentation only, and writes the information to a `REFERENCE.md` file.

Puppet Strings is based on the YARD Ruby documentation tool. To learn more about YARD, see the YARD documentation.

**Related information**

Documenting modules on page 126
Document any module you write, whether your module is for internal use only or for publication on the Forge. Complete, clear documentation helps your module users understand what your module can do and how to use it.

Puppet Strings style guide on page 137
To document your module with Puppet Strings, add descriptive tags and comments to your module code. Write consistent, clear code comments, and include at least basic information about each element of your module (such as classes or defined types).

## Install Puppet Strings

Before you can generate module documentation, you must install the Puppet Strings gem.

Puppet Strings requires:

- Ruby 2.1.9 or newer.
- Puppet 4.0 or newer.
- The `yard` Ruby gem.

1. If you don't have the `yard` gem installed yet, install it by running `gem install yard`
2. Install the `puppet-strings` gem by running `gem install puppet-strings`

## Generating documentation with strings

Generate documentation in HTML, JSON, or Markdown by running Puppet Strings.

Strings creates reference documentation based on the code and comments in all Puppet and Ruby source files in the following module subdirectories:

- `manifests/`
- `functions/`
- `lib/`
- `types/`
- `tasks/`

By default, Strings outputs HTML of the reference information and the module README to the module's `doc/` directory. You can open and read the generated HTML documentation in any browser. If you specify JSON or

Markdown output, documentation includes the reference information only. Strings writes Markdown output to a `REFERENCE.md` file and sends JSON output to `STDOUT` , but you can specify a custom file destination for Markdown and JSON output.

## Generate and view documentation in HTML

To generate HTML documentation for a Puppet module, run Strings from that module's directory.

1. Change directory into the module by running `cd /modules/<MODULE_NAME>`
2. Generate documentation with the `puppet strings` command:
   a) To generate the documentation for the entire module, run `puppet strings`
   b) To generate the documentation for specific files or directories in a module, run the `puppet strings generate` subcommand, and specify the files or directories as a space-separated list.

   For example:

   ```
   puppet strings generate first.pp second.pp
   ```

   ```
   puppet strings generate 'modules/apache/lib/**/*.rb' 'modules/apache/
   manifests/**/*.pp' 'modules/apache/functions/**/*.pp'
   ```

Strings outputs HTML to the `doc/` directory in the module. To view the generated HTML documentation for a module, open the `index.html` file in the module's `doc/` folder. To view HTML documentation for all of your local modules, run `puppet strings server` from any directory. This command serves documentation for all modules in the module path at `http://localhost:8808`. To learn more about the modulepath, see the modulepath documentation.

## Generate and view documentation in Markdown

To generate reference documentation in Markdown, specify the `markdown` format when you run Puppet Strings.

The reference documentation includes descriptions, usage details, and parameter information for classes, defined types, functions, tasks, plans, and resource types and providers.

Strings generates Markdown output as a `REFERENCE.md` file in the main module directory, but you can specify a different filename or location with command line options.

1. Change directory into the module: `cd /modules/<MODULE_NAME>`
2. Run the command: `puppet strings generate --format markdown` . To specify a different file, use the `--out` option and specify the path and filename:

   ```
   puppet strings generate --format markdown --out docs/INFO.md
   ```

View the Markdown file by opening it in a text editor or Markdown viewer.

## Generate documentation in JSON

To generate reference documentation as JSON output to a file or to standard output, specify the `json` format when you run Strings.

Generate JSON output if you want to use the documentation in a custom application that reads JSON. By default, Strings prints JSON output to `STDOUT`. For details about Strings JSON output, see the Strings JSON schema.

1. Change directory into the module: `cd /modules/<MODULE_NAME>`
2. Run the command: `puppet strings generate --format json` . To generate JSON documentation to a file instead, use the `--out` option and specify a filename:

   ```
   puppet strings generate --format json --out documentation.json
   ```

## Publish module documentation to GitHub Pages

To make your module documentation available on GitHub Pages, generate and publish HTML documentation with a Strings Rake task.

The `strings:gh_pages:update` Rake task is available in the `puppet-strings/tasks` directory. This Rake task keeps the `gh-pages` branch up to date with your current code, performing the following actions:

1. Creates a `doc` directory in the root of your project, if it doesn't already exist.
2. Creates a `gh-pages` branch of the current repository, if it doesn't already exist.
3. Checks out the `gh-pages` branch of the current repository.
4. Generates Strings HTML documentation.
5. Commits the documentation file and pushes it to the `gh-pages` branch with the `--force` flag.

To learn more about publishing on GitHub Pages, see the GitHub Pages documentation.

1. If this is the first time you are running this task, you must first update your Gemfile and Rakefile.
   a) Add the following to your Gemfile to use puppet-strings: `ruby gem 'puppet-strings'`
   b) Add the following to your Rakefile to use the `puppet-strings` tasks: `ruby require 'puppet-strings/tasks'`
2. To generate, push, and publish your module's Strings documentation, run `strings:gh_pages:update`

The documentation is published after the task pushes the updated documentation to GitHub Pages.

## Puppet Strings command reference

Modify the behavior of Puppet Strings by specifying command actions and options.

### `puppet strings` command

Generates module documentation based on code and code comments. By default, running `puppet strings` generates HTML documentation for a module into a `./doc/` directory within that module.

To pass options or arguments, such as specifying Markdown or JSON output, use the `generate` action.

Usage:

```
puppet strings [--generate] [--server]
```

| Action | Description |
|---|---|
| `generate` | Generates documentation with any specified parameters, including format and output location. |
| `server` | Serves documentation locally at `http://localhost:8808` for all modules in the modulepath. For information about the modulepath, see the modulepath documentation. |

### `puppet strings generate` action

Generates documentation with any specified parameters, including format and output location.

Usage:

```
puppet strings generate [--format <FORMAT>][--out <DESTINATION>]
 [<ARGUMENTS>]
```

For example:

```
puppet strings generate --format markdown --out docs/info.md
```

```
puppet strings generate manifest1.pp manifest2.pp
```

| Option | Description | Values | Default |
|---|---|---|---|
| `--format` | Specifies a format for documentation. | Markdown, JSON | If not specified, outputs HTML documentation. |
| `--out` | Specifies an output location for documentation. | A valid directory location and filename. | If not specified, outputs to default locations depending on format:<br>• HTML: `./doc/`<br>• Markdown: main module directory)<br>• JSON: `STDOUT` |
| Filenames or directory paths | Outputs documentation for only specified files or directories. | Valid filenames or directory paths | If not specified, outputs documentation for the entire module. |
| `--debug`, `-d` | Logs debug information. | None. | If not specified, does not log debug information. |
| `--help` | Displays help documentation for the command. | None. | If specified, returns help information. |
| `--markup <FORMAT>` | The markup format to use for documentation | • "markdown"<br>• "textile"<br>• "rdoc"<br>• "ruby"<br>• "text"<br>• "html"<br>• "none" | If no `--format` is specified, outputs HTML. |
| `--verbose`, `-v` | Logs verbosely. | None. | If not specified, logs basic information. |

#### `puppet strings server` action

Serves documentation locally at `http://localhost:8808` for all modules in the module path.

Usage:

```
puppet strings server [--markup <FORMAT>][[module_name]...][--modulepath
 <PATH>]
```

For example:

```
puppet strings server --modulepath path/to/modules
```

```
puppet strings server concat
```

| Option | Description | Values | Default |
|---|---|---|---|
| `--markup <FORMAT>` | The markup format to use for documentation | • "markdown"<br>• "textile"<br>• "rdoc"<br>• "ruby"<br>• "text"<br>• "html"<br>• "none" | If no `--format` is specified, outputs HTML. |
| `--debug`, `-d` | Logs debug information. | None. | If not specified, does not log debug information. |
| `--help` | Displays help documentation for the command. | None. | If specified, returns help information. |
| `Module name` | Generates documentation for the named module only. | A valid module name. | If not specified, generates documentation for all modules in the modulepath. |
| `--modulepath` | Puppet option for setting the modulepath. | A valid path. | Defaults to the module path specified in the `puppet.conf` file. |
| `--verbose`, `-v` | Logs verbosely. | None. | If not specified, logs basic information. |

### Available Strings tags

**`@api`**

Describes the resource as belonging to the private or public API. To mark a module element, such as a class, as private, specify as private:

```
# @api private
```

**`@example`**

Shows an example snippet of code for an object. The first line is an optional title, and any subsequent lines are automatically formatted as a code snippet. Use for specific examples of a given component. Use one example tag per example.

**`@param`**

Documents a parameter with a given name, type and optional description.

**`@!puppet.type.param`**

Documents dynamic type parameters. See the documenting resource types in the Strings style guide for detailed information.

**`@!puppet.type.property`**

Documents dynamic type properties. See the documenting resource types in the Strings style guide for detailed information.

**`@option`**

Used with a `@param` tag to defines what optional parameters the user can pass in an options hash to the method. For example:

```
  # @param [Hash] opts
```

```
  #      List of options
  # @option opts [String] :option1
  #      option 1 in the hash
  # @option opts [Array] :option2
  #      option 2 in the hash
```

**@raise**

Documents any exceptions that can be raised by the given component. For example:

```
# @raise PuppetError this error is
raised if x
```

**@return**

Describes the return value (and type or types) of a method. You can list multiple return tags for a method if the method has distinct return cases. In this case, begin each case with "if". For example:

```
# An example 4.x function.

 Puppet::Functions.create_function(:example)
 do
     # @param first The first
 parameter.
     # @param second The second
 parameter.
     # @return [String] If second
 argument is less than 10, the name
 of one item.
     # @return [Array] If second
 argument is greater than 10, a list
 of item names.
     # @example Calling the
 function.
     #   example('hi', 10)
     dispatch :example do
       param 'String', :first
       param 'Integer', :second
     end
     # ...
   end
```

**@see**

Adds "see also" references. Accepts URLs or other code objects with an optional description at the end. The URL or object is automatically linked by YARD and does not need markup formatting. Appears in the generated documentation as a "See Also" section. Use one tag per reference, such as a website or related method.

**@since**

Lists the version in which the object was first added. Strings does not verify that the specified version exists. You are responsible for providing accurate information.

**@summary**

A description of the documented item, of 140 characters or fewer.

# Puppet Strings style guide

To document your module with Puppet Strings, add descriptive tags and comments to your module code. Write consistent, clear code comments, and include at least basic information about each element of your module (such as classes or defined types).

Strings uses YARD-style tags and comments, along with the structure of the module code, to generate complete reference information for your module. Whenever you update your code, update your documentation comments at the same time.

This style guide applies to:

- Puppet Strings version 2.0 or later
- Puppet 4.0 or later

For information about the specific meaning of the terms 'must,' 'must not,' 'required,' 'should,' 'should not,' 'recommend,' 'may,' and 'optional,' see RFC 2119.

## The module README

In your module README, include basic module information and extended usage examples for common use cases. The README tells users what your module does and how to use it. Strings generates reference documentation, so typically, there is no need to include a reference section in your README. However, Strings does not generate information for type aliases or facts; if your module includes these elements, include a short reference section in your README with information about these elements only.

Include the following sections in the README:

| | |
|---|---|
| **Module description** | What the module does and why it is useful. |
| **Setup** | Prerequisites for module use and getting started information. |
| **Usage** | Instructions and examples for common use cases or advanced configuration options. |
| **Reference** | Only if the module contains facts or type aliases, include a short Reference section. Other reference information is handled by Strings, so don't repeat it in the README. |
| **Limitations** | Operating system compatibility and known issues. |
| **Development** | Guidelines for contributing to the module |

## Comment style guidelines

Strings documentation comments inside module code follow these rules and guidelines:

- Place an element's documentation comment immediately before the code for that element. Do not put a blank line between the comment and its corresponding code.
- Each comment tag (such as `@example)` may have more than one line of comments. Indent additional lines with two spaces.
- Keep each comment line to no more than 140 characters, to improve readability.
- Separate comment sections (such as `@summary`, `@example`, or the `@param` list) with a blank comment line (that is, a # with no additional content), to improve readability.
- Untagged comments for a given element are output in an overview section that precedes all tagged information for that code element.
- If an element, such as a class or parameter, is deprecated, indicate it in the description for that element with **Deprecated** in bold.

### Classes and defined types

Document each class and defined type, along with its parameters, with comments before the code. List the class and defined type information in the following order:

1. A `@summary` tag, a space, and then a summary describing the class or defined type.
2. Other tags such as `@see`, `@note`, or `@api private`.
3. Usage examples, each consisting of:
   a. An `@example` tag with a description of a usage example on the same line.
   b. A code example showing how the class or defined type is used. Place this example directly under the `@example` tag and description, indented two spaces.
4. One `@param` tag for each parameter in the class or defined type. See the parameters section for formatting guidelines.

### Parameters

Add parameter information as part of any class, defined type, or function that accepts parameters. Include the parameter information in the following order:

1. The `@param` tag, a space, and then the name of the parameter.
2. A description of what the parameter does. This may be either on the same line as the `@param` tag or on the next line, indented with two spaces.
3. Additional information about valid values that is not clear from the data type. For example, if the data type is `[String]`, but the value must specifically be a path, say so here.
4. Other information about the parameter, such as warnings or special behavior. For example:

```
# @param noselect_servers
#   Specifies one or more peers to not sync with. Puppet appends
 'noselect' to each matching item in the `servers` array.
```

### Example class

```
# @summary configures the Apache PHP module
#
# @example Basic usage
#   class { 'apache::mod::php':
#     package_name => 'mod_php5',
#     source       => '/etc/php/custom_config.conf',
#     php_version  => '7',
#   }
#
# @see http://php.net/manual/en/security.apache.php
#
# @param package_name
#   Names the package that installs mod_php
# @param package_ensure
#   Defines ensure for the PHP module package
# @param path
#   Defines the path to the mod_php shared object (.so) file.
# @param extensions
#   Defines an array of extensions to associate with PHP.
# @param content
#   Adds arbitrary content to php.conf.
# @param template
#   Defines the path to the php.conf template Puppet uses to generate the
 configuration file.
# @param source
#   Defines the path to the default configuration. Values include a
 puppet:/// path.
```

```
# @param root_group
#   Names a group with root access
# @param php_version
#   Names the PHP version Apache is using.
#
class apache::mod::php (
  $package_name     = undef,
  $package_ensure   = 'present',
  $path             = undef,
  Array $extensions = ['.php'],
  $content          = undef,
  $template         = 'apache/mod/php.conf.erb',
  $source           = undef,
  $root_group       = $::apache::params::root_group,
  $php_version      = $::apache::params::php_version,
)
 {
 …
 }
```

**Example defined type**

```
# @summary
#   Create and configure a MySQL database.
#
# @example Create a database
#   mysql::db { 'mydb':
#     user     => 'myuser',
#     password => 'mypass',
#     host     => 'localhost',
#     grant    => ['SELECT', 'UPDATE'],
#   }
#
# @param name
#   The name of the database to create. (dbname)
# @param user
#   The user for the database you're creating.
# @param password
#   The password for $user for the database you're creating.
# @param dbname
#   The name of the database to create.
# @param charset
#   The character set for the database.
# @param collate
#   The collation for the database.
# @param host
#   The host to use as part of user@host for grants.
# @param grant
#   The privileges to be granted for user@host on the database.
# @param sql
#   The path to the sqlfile you want to execute. This can be single file
 specified as string, or it can be an array of strings.
# @param enforce_sql
#   Specifies whether to execute the sqlfiles on every run. If set to false,
 sqlfiles  runs only one time.
# @param ensure
#   Specifies whether to create the database. Valid values are 'present',
 'absent'. Defaults to 'present'.
# @param import_timeout
#   Timeout, in seconds, for loading the sqlfiles. Defaults to 300.
# @param import_cat_cmd
```

```
#   Command to read the sqlfile for importing the database. Useful for
 compressed sqlfiles. For example, you can use 'zcat' for .gz files.
```

#### Functions

For custom Ruby functions, place documentation strings immediately before each dispatch call. For functions written in Puppet, place documentation strings immediately before the function name.

Include the following information for each function:

1. An untagged docstring describing what the function does.
2. One `@param` tag for each parameter in the function. See the parameters section for formatting guidelines.
3. A `@return` tag with the data type and a description of the returned value.
4. Optionally, a usage example, consisting of:
   a. An `@example` tag with a description of a usage example on the same line.
   b. A code example showing how the function is used. Place this example directly under the `@example` tag and description, indented two spaces.

#### Example Ruby function with one potential return type

```
# An example 4.x function.
Puppet::Functions.create_function(:example) do
  # @param first The first parameter.
  # @param second The second parameter.
  # @return [String] Returns a string.
  # @example Calling the function
  #   example('hi', 10)
  dispatch :example do
    param 'String', :first
    param 'Integer', :second
  end

  # ...
end
```

#### Example Ruby function with multiple potential return types

If the function has more than one potential return type, specify a `@return` tag for each. Begin each tag string with "if" to differentiate between cases.

```
# An example 4.x function.
Puppet::Functions.create_function(:example) do
  # @param first The first parameter.
  # @param second The second parameter.
  # @return [String] If second argument is less than 10, the name of one
 item.
  # @return [Array] If second argument is greater than 10, a list of item
 names.
  # @example Calling the function.
  #   example('hi', 10)
  dispatch :example do
    param 'String', :first
    param 'Integer', :second
  end

  # ...
end
```

#### Puppet function example

```
# @param name the name to say hello to.
# @return [String] Returns a string.
# @example Calling the function.
#    example('world')
function example(String $name) {
    "hello, $name"
}
```

#### Resource types

Add descriptions to the type and its attributes by passing either a here document (or "heredoc") or a short string to the `desc` method.

Strings automatically detects much of the information for types, including the parameters and properties, collectively known as attributes. To document the resource type itself, pass a heredoc to the `desc` method immediately after the type definition. Using a heredoc allows you to use multiple lines and Strings comment tags for your type documentation. For details about heredocs in Puppet, see the topic about heredocs in the language reference.

For attributes, where a short description is usually enough, pass a string to `desc` in the attribute. As with the `@param` tag, keep descriptions to 140 or fewer characters. If you need a longer description for an attribute, pass a heredoc to `desc` in the attribute itself.

You do not need to add tags for other method calls. Every other method call present in a resource type is automatically included and documented by Strings, and each attribute is updated accordingly in the final documentation. This includes method calls such as `defaultto`, `newvalue`, and `namevar`. If your type dynamically generates attributes, document those attributes with the `@!puppet.type.param` and `@!puppet.type.property` tags before the type definition. You may not use any other tags before the resource type definition.

Document the resource type description in the following order:

1. Directly under the type definition, indented two spaces, the `desc` method, with a heredoc including a descriptive delimiting keyword, such as `DESC`.
2. A `@summary` tag with a summary describing the type.
3. Optionally, usage examples, each consisting of:
    a. An `@example` tag with a description of a usage example on the same line.
    b. Code example showing how the type is used. Place this example directly under the `@example` tag and description, indented two spaces.

For types created with the resource API, follow the guidelines for standard resource types, but pass the heredoc or documentation string to a `desc` key in the data structure. You can include tags and multiple lines with the heredoc. Strings extracts the heredoc information along with other information from this data structure.

#### Example resource API type

The heredoc and documentation strings that Strings uses are called out in bold in this code example:

```
Puppet::ResourceApi.register_type(
  name: 'apt_key',
  docs: <<-EOS,
@summary Fancy new type.
@example Fancy new example.
 apt_key { '6F6B15509CF8E59E6E469F327F438280EF8D349F':
   source => 'http://apt.puppetlabs.com/pubkey.gpg'
 }

This type provides Puppet with the capabilities to
manage GPG keys needed by apt to perform package validation. Apt has its own
 GPG keyring that can be manipulated through the `apt-key` command.
```

```
**Autorequires**:
If Puppet is given the location of a key file which looks like an absolute
 path this type will autorequire that file.
EOS
  attributes:   {
    ensure:      {
      type: 'Enum[present, absent]',
       desc: 'Whether this apt key should be present or absent on the target
 system.'**
    },
    id:          {
      type:      'Variant[Pattern[/\A(0x)?[0-9a-fA-F]{8}\Z/], Pattern[/
\A(0x)?[0-9a-fA-F]{16}\Z/], Pattern[/\A(0x)?[0-9a-fA-F]{40}\Z/]]',
      behaviour: :namevar,
       desc:      'The ID of the key you want to manage.',**
    },
    # ...
    created:     {
      type:      'String',
      behavior: :read_only,
      desc:      'Date the key was created, in ISO format.',**
    },
  },
  autorequires: {
    file:    '$source', # will evaluate to the value of the `source`
 attribute
    package: 'apt',
  },
)
```

### Puppet tasks and plans

Strings documents Puppet tasks automatically, taking all information from the task metadata. Document task plans just as you would a class or defined type, with tags and descriptions in the plan file.

List the plan information in the following order:

1. A `@summary` tag, a space, and then a summary describing the plan.
2. Other tags such as `@see`, `@note`, or `@api private`.
3. Usage examples, each consisting of:
    a. An `@example` tag with a description of a usage example on the same line.
    b. Code example showing how the plan is used. Place this example directly under the `@example` tag and description, indented two spaces.
4. One `@param` tag for each parameter in the plan. See the parameters section for formatting guidelines. For example:

```
# @summary A simple plan.
#
# @param param1
#   First parameter description.
# @param param2
#   Second parameter description.
# @param param3
#   Third parameter description.
plan mymodule::my_plan(String $param1, $param2, Integer $param3 = 1) {
  run_task('mymodule::lb_remove', $param1, target => $param2)
}
```

# Publishing modules

To share your module with other Puppet users, get contributions to your modules, and maintain your module releases, publish your module on the Puppet Forge. The Forge is a community repository of modules, written and contributed by open source Puppet and Puppet Enterprise users.

To publish your module, you'll:

1. Create a Forge account, if you don't already have one.
2. Prepare your module for packaging.
3. Add module metadata in the `metadata.json` file.
4. Build an uploadable tarball of your module.
5. Upload your module using the Forge web interface.

### Naming your module

Your module has two names: a short name, like "mysql", and a long name that includes your Forge username, like "puppetlabs-mysql". When you upload your module to the Forge, use the module's long name.

Your module's short name is the same as that module's directory on your disk. This name must consist of letters, numbers, and underscores only; it can't contain dashes or periods.

The long name is composed of your Forge username and the short name of your module. For example, the "puppetlabs" user maintains a "mysql" module, which is located in a `./modules/mysql` directory and is known to the Forge as "puppetlabs-mysql".

In your module's `metadata.json` file, always use the long name of your module. This helps disambiguate modules that might have common short names, such as "mysql" or "apache." If you created your module with Puppet Development Kit (PDK), and you provided your Forge username to PDK, the `metadata.json` file already contains the correct long name for the module. Otherwise, edit your module's metadata with the correct long name.

**Tip:** Although the Forge expects to receive modules named `username-module`, its web interface presents them as `username/module`. Always use the `username-module` style in your metadata files and when issuing commands.

### Related information

Module metadata on page 120
Your module must contain certain metadata in a `metadata.json` file, which tracks important information about the module.

## Create a Forge account

To publish your modules to the Forge, you must first create a Forge account.

1. In your web browser, navigate to the Forge website and click **Sign Up**.
2. Fill in the fields on the sign-up form. The username you pick becomes part of your module long name, such as "bobcat-apache".
3. Check your email for a verification email from the Forge, and then follow the instructions in the email to verify your email address.

After you have verified your email address, you can publish modules to the Forge.

## Prepare your module for publishing

Before you build your module package for publishing, make sure it's ready to be packaged.

Exclude unnecessary files from your package, remove or ignore any symlinks your module contains, and make sure your `metadata.json` file contains the correct information.

**Tip:** To publish your module to the Forge, your README, license file, changelog, and `metadata.json` must be UTF-8 encoded. If you used Puppet Development Kit (or the deprecated `puppet module generate` command) to create your module, these files are already UTF-8 encoded.

### Excluding files from the package

To exclude certain files from your module build, include them in either an ignore file. Ignore files are useful for excluding files that are not needed to run the module, such as temporary files or files generated by spec tests. The ignore file must be in the root directory. You can use `.pdkignore`, `.gitignore`, or `.pmtignore` files in your module.

If you are building your module with PDK, your module package contains a `.pdkignore` file that already includes a list of commonly ignored files. To add or remove files to this list, define them in the module's `.sync.yml` file. For more information about customizing your module's configuration with `.sync.yml`, see the PDK documentation.

If you are building your module with the `puppet module build` command, create a `.pmtignore` file and in it, list the files you want to exclude from the module package.

To prevent files, such as those in temporary directories, from ever being checked into your module's Git repo, list the files in a `.gitignore` file.

For example, a typical ignore file might look like this:

```
import/
/spec/fixtures/
.tmp
*.lock
*.local
.rbenv-gemsets
.ruby-version
build/
docs/
tests/
log/
junit/
tmp/
```

### Removing symlinks from your module

Symlinks in modules are unsupported. If your module contains symlinks, either remove them or ignore them before you build your module.

If you try to build a module package that contains symlinks, you receive the following error:

```
Warning: Symlinks in modules are unsupported. Please investigate symlink
 manifests/my-module.pp->manifests/init.pp.
Error: Found symlinks. Symlinks in modules are not allowed, please remove
 them.
Error: Try 'puppet help module build' for usage
```

### Verifying metadata

To publish your module on the Forge, it must contain required metadata in a `metadata.json` file. If you created your module with PDK or the deprecated `puppet module generate` command, you'll already have a `metadata.json` file. Open the file in any text editor, and make any necessary edits. For details on writing or editing the `metadata.json` file, see the related topic about module metadata.

## Build a module package

To upload your module to the Forge, first build an uploadable module package with Puppet Development Kit.

PDK builds a `.tar.gz` package with the naming convention `<USERNAME>-<MODULE_SHORT_NAME>-<VERSION>.tar.gz`. in the module's `pkg/` subdirectory. For complete details about this task, see the PDK topic about building module packages.

1. Change into the module directory by running `cd <MODULE_DIRECTORY>`
2. Build the package by running `pdk build`
3. Answer the question prompts as needed. You can use default answers to optional questions by pressing **Enter** at the prompt.
4. At the confirmation prompt, confirm or cancel package creation.

## Upload a module to the Forge

To publish a new module release to the Forge, upload the module tarball using the web interface.

The module package must be a compiled `tar.gz` package of 10MB or less.

1. In your web browser, navigate to the Puppet Forge and log in.
2. Click **Publish** in the upper right hand corner of the screen.
3. On the upload page, click **Choose File** and use the file browser to locate and select the release tarball. Then click **Upload Release**.

After a successful upload, your browser loads the new release page for your module. If there were any errors on your upload, they appear on the same screen. Your module's README, Changelog, and License files are displayed on your module's Forge page.

### Publish modules to the Forge with Travis CI

You can automatically publish new versions of your module to the Forge using Travis CI.

1. If this is your first time using Travis CI for automatic publishing, you must first enable Travis CI to publish to the Forge.
   a) Enable Travis CI for the module repository.
   b) Generate a Travis-encrypted Forge password string. For instructions, see the Travis CI encryption keys docs.
   c) Create a `.travis.yml` file in the module's repository base. Include a deployment section that includes your Forge username and the encrypted Forge password, such as:

   ```
   deploy:
     provider: puppetforge
     user: <FORGE_USER>
     password:
       secure: "<ENCRYPTED_FORGE_PASSWORD>"
     on:
       tags: true
       # all_branches is required to use tags
       all_branches: true
   ```
2. To publish to the Forge with Travis CI, update, tag, and push your repository.
   a) Update the version number in the module's `metadata.json` file and commit the change to the module repository.
   b) Tag the module repo with the desired version number. For more information about how to do this, see Git docs on basic tagging.
   c) Push the commit and tag to your Git repository. Travis CI builds and publish the module.

## Deprecate a module on the Forge

To let your module users know that you are no longer maintaining your module and that they should stop using it, deprecate your module on the Forge.

File a ticket in the FORGE project on the Puppet JIRA site. The ticket should include:

- The full name of the module to be deprecated, such as `puppetlabs-apache`.
- The reason for the deprecation. The reason is publicly displayed on the Forge.
- A recommended alternative module or workaround.

## Delete a module release from the Forge

To delete a release of your module, use the Forge web interface. A deleted release is still downloadable via the Forge page or `puppet module` command if a user requests the module by specific version.

**Restriction:** You cannot delete a released version and then upload a new version of the same release.

1. In your web browser, navigate to the Puppet Forge and log in.
2. Click **Your Modules**.
3. Go to the page of the module release you want to delete.
4. Click **Select another release**, choose the release you want from the drop-down list, and click **Delete**.
5. On the confirmation page that loads, supply a reason for the deletion and submit.

   The reason you give for deleting your module is visible to Forge users.
6. Click **Yes, delete it**.

On your module page, the confirmation of the deletion is displayed.

### Related information

Finding and downloading deleted modules on page 108
You can still search for and download a specific release of a module on the Forge, even if the release has been deleted.

# Contributing to Puppet modules

Contribute to Puppet modules to help add new functionality, fix bugs, or make other improvements.

Your contributions help us serve a greater spectrum of platforms, hardware, software, and deployment configurations. We appreciate all kinds of user contributions, including:

- Bug reports.
- Feature requests.
- Participation in our community discussion group or chat.
- Code changes, such as bug fixes or new functionality.
- Documentation changes, such as corrections or new usage examples.
- Reviewing pull requests.

To make bug reports or feature requests, create a JIRA ticket in the Puppet MODULES project. If you are requesting a feature, describe the use case for it and the goal of the feature. If you are filing a bug report, clearly describe the problem and the steps to reproduce it.

Participating in community discussions is a great way to get involved. Join the community conversations in the puppet-users discussion group or our community Slack chat:

- To join the discussion group, see the puppet-users Google group.
- To join our community chat, see the Puppet Community Slack.

We ask everyone participating in Puppet communities to abide by our code of conduct. See our community guidelines page for details.

## Contributing changes to module repositories

To contribute bug fixes, new features, expanded functionality, or documentation to Puppet modules, submit a pull request to our module repositories on GitHub.

When working on Puppet modules, follow this basic workflow:

1. Discuss your change with the Puppet community.
2. Fork the repository on GitHub.
3. Make changes on a topic branch of your fork, documenting and testing your changes.
4. Submit changes as a pull request to the Puppet repository.
5. Respond to any questions or feedback on your pull request.

### Before submitting a pull request

- To submit code changes, you must have a GitHub account. If you don't already have an account, sign up on GitHub.
- Know what Git best practices we use and expect.
- Sign our Contributor License Agreement.

### Discussing your change with the community

We love when people submit code changes to our projects, and we appreciate bug and typo fixes as much as we appreciate major features.

If you are proposing a significant or complex change to a Puppet module, we encourage you to discuss potential changes and their impact with the Puppet community.

To propose and discuss a change, send a message to the puppet-users discussion group or bring it up in the Puppet Community Slack #forge-modules channel.

### Forking the repository and creating a topic branch

Fork the repository you want to make changes to, and create a topic branch for your work.

Give your topic branch a name that describes the work you're contributing. Always base the topic branch on the repository's master branch, unless one of our module developers specifically asks you to base it on a different branch.

**Remember:** Never work directly on the master branch or any other core branch.

### Making changes

When you make changes to a Puppet module, make changes that are compatible with all currently supported versions of Puppet. Do not break users' existing installations or configurations with your changes. For a list of supported versions, see the Puppet component version page.

To add new classes, defined types, or tasks to a module, use Puppet Development Kit (PDK). PDK creates manifests and test templates, validates, and runs unit tests on your changes.

If you make a backward-incompatible change, you must include a deprecation warning for the old functionality, as well as documentation that tells users how to migrate to the new functionality. If you aren't sure how to proceed, ask for help in the puppet-users group or the community Slack chat.

### Documenting changes

When you add documentation to modules, follow our documentation style and formatting guidelines. These guidelines help make our docs clear and easier to translate into other languages.

If you make code changes to modules, you must document your changes. We can't merge undocumented changes.

To provide usage examples, add them to the README's usage section. Include information about what the user can accomplish with each usage example.

Add reference information, such as class descriptions and parameters, as Puppet Strings-compatible code comments, so that we can generate complete documentation before we release the new version of the module. Do not manually edit generated `REFERENCE.md` files; any changes you make are overwritten when we generate a new file. For complete information about writing good module documentation, see Documenting modules.

In Puppet module documentation, adhere to the following conventions:

- Lowercase module names, such as `apache`. This helps differentiate the module from the software the module is managing. When talking about the software being managed, capitalize names as they would normally be capitalized, such as `Apache`.
- Set string values in single quotes, to make it clear that they are strings. For example, `'string'` or `'C:/user/documents/example.txt'`.
- Set the values `true`, `false`, and `undef` in backticks, such as ``` `true` ```.
- Set data types in backticks, such as ``` `Boolean` ```.
- Set filenames, settings, directories, classes, types, defined types, functions, and similar code elements in backticks, unless the user passes them as a string value. If the user passes the value as a string, use quotes to make that clear.
- Do not use any special marking for integer values, such as `1024`.
- Use empty lines between new lines to help with readability.

### Testing your changes

Before you submit a pull request, make sure that you have added tests for your changes.

If you create new classes or defined types, PDK creates basic tests templates for you. Use PDK to validate and run unit tests on the module, to ensure that your changes don't accidentally break anything.

If you need further help writing tests or getting tests to work, ask for help in the puppet-user discussion group, in our community Slack chat, or if you created a JIRA ticket regarding your change, in the ticket.

If you don't know how to write tests for your changes, clearly say so in your pull request. We don't necessarily reject pull requests without tests, but someone needs to add the tests before we can merge your contribution.

### Committing your changes

As you add code, commit your work for one function at a time. Ensure the code for each commit does only one thing. This makes it easier to remove one commit and accept another, if necessary. We would rather see too many commits than too few.

In your commit message, provide:

1. A brief description of the behavior before your changes.
2. Why that behavior was a problem.
3. How your changes fix the problem.

For example, this commit message is for adding to the CONTRIBUTING document:

```
Make the example in CONTRIBUTING concrete

Without this patch applied, there is no example commit message in the
 CONTRIBUTING document. The contributor is left to imagine what the commit
 message should look. This patch adds a more specific example.
```

### Submitting changes

Submit your changes as a pull request to the `puppetlabs` organization repository on GitHub.

Push your changes to the topic branch in your fork of the repository. Submit a pull request to the `puppetlabs` repository for the module.

Someone with the permissions to merge and commit to the Puppet repository (a committer) checks whether the pull request meets the following requirements:

- It is on its own correctly named branch.
- It contains only commits relevant to the specific issue.
- It has clear commit messages that describe the problem and the solution.
- It is appropriately and clearly documented.

### Responding to feedback

Be sure to respond to any questions or feedback you receive from the Puppet modules team on your pull request. Puppet community members might also make comments or suggestions that you want to consider.

When making changes to your pull request, push your commits to the same topic branch you used for your pull request. When you push changes to the branch, it automatically updates your pull request. After the team has approved your request, someone from the modules team merges it, and your changes are included in the next release of the module.

If you do not respond to the modules team's requests, your pull request might be rejected or closed. To address such comments or questions later, create a new pull request.

## Reviewing community pull requests

As a Puppet community member, you can offer feedback on someone else's contributed code.

When reviewing pull requests, any of the following contributions are helpful:

- Review the code for any obvious problems.
- Provide feedback based on personal experience on the subject.
- Test relevant examples on an untested platform.
- Look at potential side effects of the change.
- Examine discrepancies between the original issue and the pull request.

Add your comments and questions to the pull request, pointing out any specific lines that need attention. Be sure to respond to any questions the contributor has about your comments.

# Puppet services and tools

Puppet provides a number of core services and administrative tools to manage systems with or without a Puppet master, and to compile configurations for Puppet agents.

- Puppet commands on page 150

Puppet's command line interface (CLI) consists of a single `puppet` command with many subcommands.

- Running Puppet commands on Windows on page 152

Puppet was originally designed to run on *nix systems, so its commands generally act the way *nix admins expect. Because Windows systems work differently, there are a few extra things to keep in mind when using Puppet commands.

- Puppet agent on *nix systems on page 156

Puppet agent is the application that manages the configurations on your nodes. It requires a Puppet master to fetch configuration catalogs from.

- Puppet agent on Windows on page 159

Puppet agent is the application that manages configurations on your nodes. It requires a Puppet master server to fetch configuration catalogs from.

- Puppet apply on page 163

Puppet apply is an application that compiles and manages configurations on nodes. It acts like a self-contained combination of the Puppet master and Puppet agent applications.

- Puppet device on page 165

With Puppet device, you can manage network devices, such as routers, switches, firewalls, and Internet of Things (IOT) devices, without installing a Puppet agent on them. Devices that cannot run Puppet applications require a

Puppet agent to act as a proxy. The proxy manages certificates, collects facts, retrieves and applies catalogs, and stores reports on behalf of a device.

# Puppet commands

Puppet's command line interface (CLI) consists of a single `puppet` command with many subcommands.

Puppet Server and Puppet's companion utilities Facter and Hiera , have their own CLI.

## Puppet agent

Puppet agent is a core service that manages systems, with the help of a Puppet master. It requests a configuration catalog from a Puppet master server, then ensures that all resources in that catalog are in their desired state.

For more information, see:

- Overview of Puppet's architecture
- Puppet Agent on *nix systems
- Puppet Agent on Windows systems
- Puppet Agent's man page

## Puppet Server

Using Puppet code and various other data sources, Puppet Server compiles configurations for any number of Puppet agents.

Puppet Server is a core service and has its own subcommand, `puppetserver`, which isn't prefaced by the usual `puppet` subcommand.

For more information, see:

- Overview of Puppet's architecture
- Puppet Server
- Puppet Server subcommands
- Puppet Master's man page

## Puppet apply

Puppet apply is a core command that manages systems without contacting a Puppet master server. Using Puppet modules and various other data sources, it compiles its own configuration catalog, and then immediately applies the catalog.

For more information, see:

- Overview of Puppet's architecture
- Puppet apply
- Puppet apply's man page

## Puppet ssl

Puppet ssl is a command for managing SSL keys and certificates for Puppet SSL clients needing to communicate with your Puppetinfrastructure.

Puppet ssl usage: `puppet ssl <action> [--certname <name>]`

Possible actions:

- `submit request`: Generate a certificate signing request (CSR) and submit it to the CA. If a private and public key pair already exist, they are used to generate the CSR. Otherwise, a new key pair is generated. If a CSR has already been submitted with the given `certname,` then the operation fails.

- `download_cert`: Download a certificate for this host. If the current private key matches the downloaded certificate, then the certificate is saved and used for subsequent requests. If there is already an existing certificate, it is overwritten.
- `verify`: Verify that the private key and certificate are present and match. Verify the certificate is issued by a trusted CA, and check the revocation status

### Puppet module

Puppet module is a multi-purpose administrative tool for working with Puppet modules. It can install and upgrade new modules from the Puppet Forge, help generate new modules, and package modules for public release.

For more information, see:

- Module fundamentals
- Installing modules
- Publishing modules on the Puppet Forge
- Puppet Module's man page

### Puppet resource

Puppet resource is an administrative tool that lets you inspect and manipulate resources on a system. It can work with any resource type Puppet knows about. For more information, see Puppet Resource's man page.

### Puppet config

Puppet config is an administrative tool that lets you view and change Puppet settings.

For more information, see:

- About Puppet's settings
- Checking values of settings
- Editing settings on the command line
- Short list of important settings
- Puppet Config's man page

### Puppet parser

Puppet parser lets you validate Puppet code to make sure it contains no syntax errors. It can be a useful part of your continuous integration toolchain. For more information, see Puppet Parser's man page.

### Puppet help and Puppet man

Puppet help and Puppet man can display online help for Puppet's other subcommands.

For more information, see:

- Puppet help's man page
- Puppet man's man page

### Full list of subcommands

For a full list of Puppet subcommands, see Puppet's subcommands.

# Running Puppet commands on Windows

Puppet was originally designed to run on *nix systems, so its commands generally act the way *nix admins expect. Because Windows systems work differently, there are a few extra things to keep in mind when using Puppet commands.

## Supported commands

Not all Puppet commands work on Windows. Notably, Windows nodes can't run the `puppet master` or `puppet cert` commands.

The following commands are designed for use on Windows:

- `puppet agent`
- `puppet apply`
- `puppet module`
- `puppet resource`
- `puppet config`
- `puppet lookup`
- `puppet help`
- `puppet man`

## Running Puppet's commands

The installer adds Puppet commands to the PATH. After installing, you can run them from any command prompt (cmd.exe) or PowerShell prompt.

Open a new command prompt after installing. Any processes that were already running before you ran the installer do not pick up the changed PATH value.

## Running with administrator privileges

You usually want to run Puppet's commands with administrator privileges.

Puppet has two privilege modes:

- Run with limited privileges, only manage certain resource types, and use a user-specific `confdir` and codedir
- Run with administrator privileges, manage the whole system, and use the system `confdir` and codedir

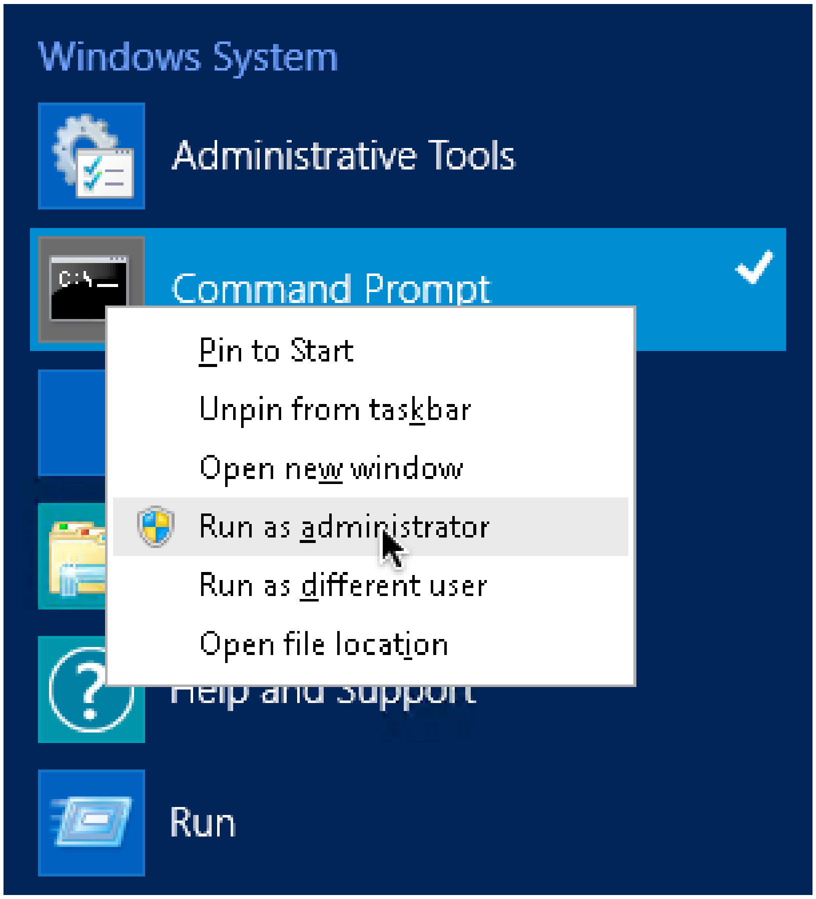
On *nix systems, Puppet defaults to running with limited privileges, when not run by `root`, but can have its privileges raised with the standard sudo command.

Windows systems don't use sudo, so escalating privileges works differently.

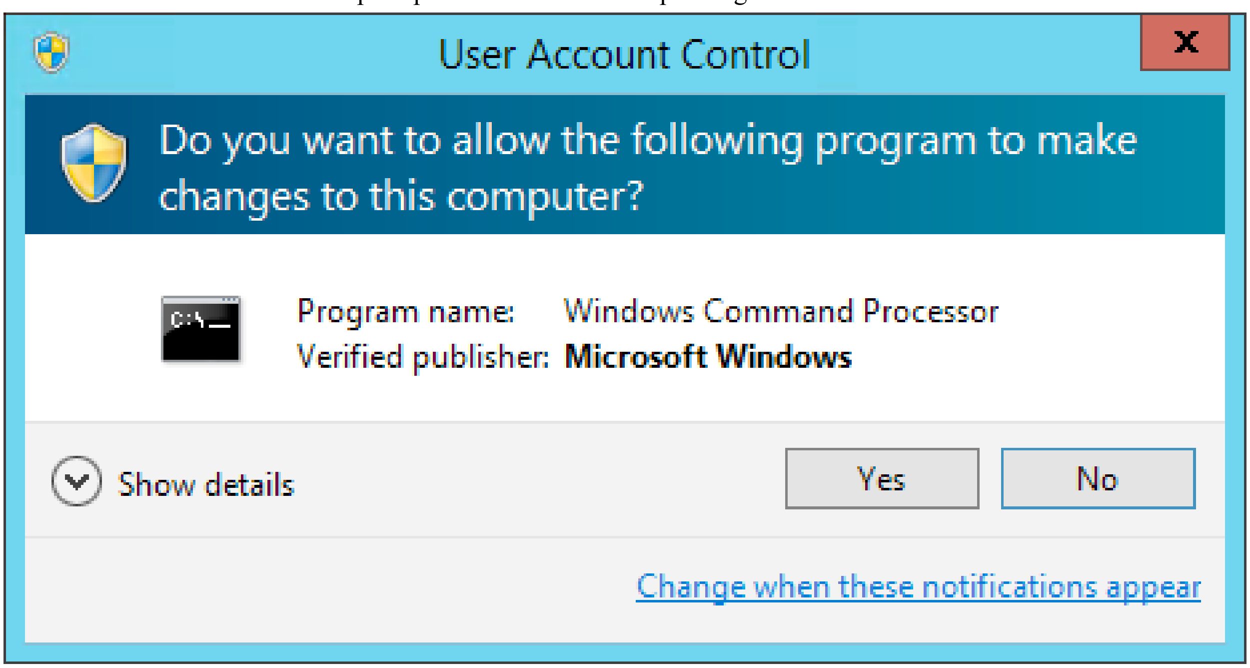
Newer versions of Windows manage security with User Account Control (UAC), which was added in Windows 2008 and Windows Vista. With UAC, most programs run by administrators still have limited privileges. To get administrator privileges, the process has to request those privileges when it starts.

To run Puppet's commands in adminstrator mode, you must first start a Powershell command prompt with administrator privileges.

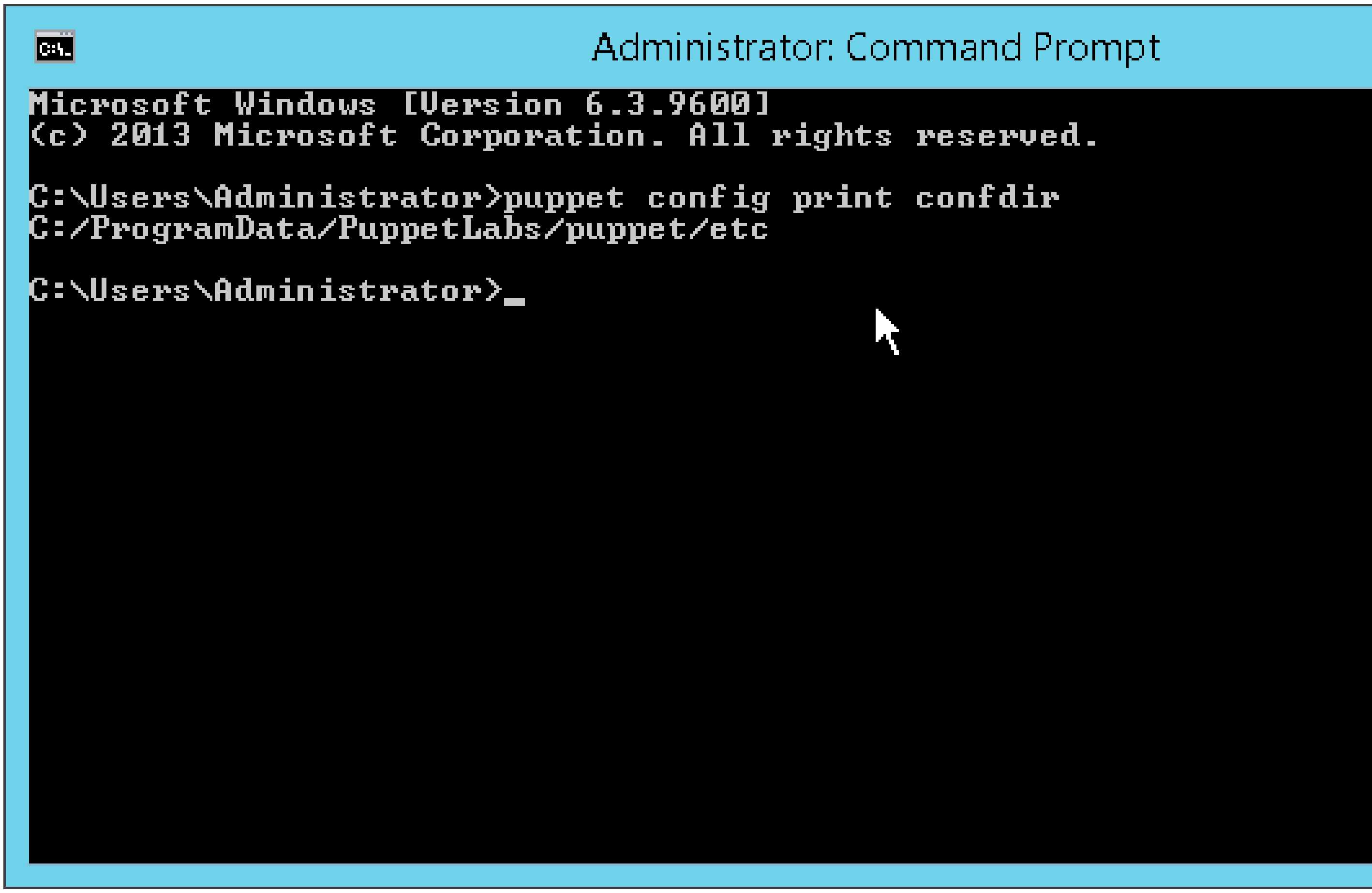
Right-click the **Start** (or apps screen tile) -> **Run as administrator**:




Click **Yes** to allow the command prompt to run with elevated privaleges:

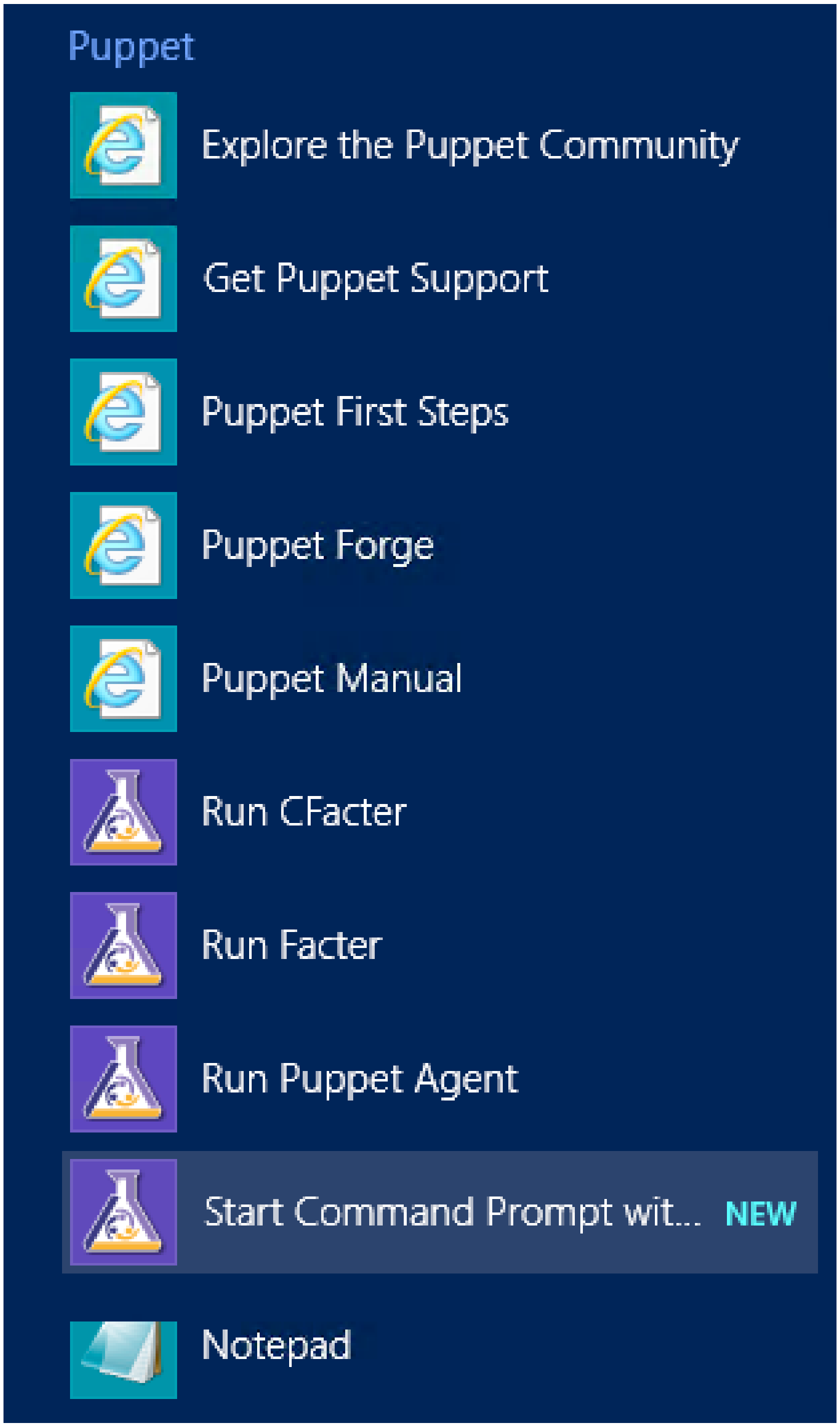



The title bar on the comand prompt window begins with **Administrator**. This means Puppet commands that run from that window can manage the whole system.


Administrator: Command Prompt
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.
C:\Users\Administrator>puppet config print confdir
C:/ProgramData/PuppetLabs/puppet/etc
C:\Users\Administrator>_

## The Puppet Start menu items

Puppet’s installer adds a folder of shortcut items to the **Start** Menu.




These items aren’t necessary for working with Puppet, because Puppet agent runs a normal Windows service and the Puppet commands work from any command or PowerShell prompt. They’re provided solely as conveniences.

The **Start** menu items do the following:

| | |
|---|---|
| **Run Facter** | This shortcut requests UAC elevation and, using the CLI, runs Facter with administrator privileges. |

| | |
|---|---|
| **Run Puppet agent** | This shortcut requests UAC elevation and, using the CLI, performs a single Puppet agent command with administrator privileges. |
| **Start Command Prompt with Puppet** | This shortcut starts a normal command prompt with the working directory set to Puppet's program directory. The CLI window icon is also set to the Puppet logo. This shortcut was particularly useful in previous versions of Puppet, before Puppet's commands were added to the PATH at installation time.<br><br>**Note:** This shortcut does not automatically request UAC elevation; just like with a normal command prompt, you'll need to right-click the icon and choose **Run as administrator**. |

### Configuration settings

Configuration settings can be viewed and modified using the CLI.

To get configuration settings, run: `puppet agent --configprint <SETTING>`

To set configuration settings, run: `puppet config set <SETTING VALUE> --section <SECTION>`

When running Puppet commands on Windows, note the following:

- The location of `puppet.conf` depends on whether the process is running as an administrator or not.
- Specifying file owner, group, or mode for file-based settings is not supported on Windows.
- The `puppet.conf` configuration file supports Windows-style CRLF line endings as well as *nix-style LF line endings. It does not support Byte Order Mark (BOM). The file encoding must either be UTF-8 or the current Windows encoding, for example, Windows-1252 code page.
- Common configuration settings are `certname`, `server`, and `runinterval`.
- You must restart the Puppet agent service after making any changes to Puppet’s `runinterval` config file setting.

## Puppet master

## Puppet agent on *nix systems

Puppet agent is the application that manages the configurations on your nodes. It requires a Puppet master to fetch configuration catalogs from.

Depending on your infrastructure and needs, you can manage systems with Puppet agent as a service, as a cron job, or on demand.

For more information about running the `puppet agent` command, see the puppet agent man page.

### Puppet agent's run environment

Puppet agent runs as a specific user, (usually `root`) and initiates outbound connections on port 8140.

#### Ports

Puppet’s HTTPS traffic uses port 8140. Your operating system and firewall must allow Puppet agent to initiate outbound connections on this port.

If you want to use a non-default port, you have to change the masterport setting on all agent nodes, and ensure that you change your Puppet master’s port as well.

### User

Puppet agent runs as `root`, which lets it manage the configuration of the entire system.

Puppet agent can also run as a non-root user, as long as it is started by that user. However, this restricts the resources that Puppet agent can manage, and requires you to run Puppet agent as a cron job instead of a service.

If you need to install packages into a directory controlled by a non-root user, use an `exec` to unzip a tarball or use a recursive `file` resource to copy a directory into place.

When running without root permissions, most of Puppet's resource providers cannot use `sudo` to elevate permissions. This means Puppet can only manage resources that its user can modify without using `sudo`.

Out of the core resource types listed in the resource type reference, only the following types are available to non-root agents:

| Resource type | Details |
|---|---|
| `augeas` | |
| `cron` | Only non-root cron jobs can be viewed or set. |
| `exec` | Cannot run as another user or group. |
| `file` | Only if the non-root user has read/write privileges. |
| `notify` | |
| `schedule` | |
| `service` | For services that don't require root. You can also use the `start`, `stop`, and `status` attributes to specify how non-root users should control the service. |
| `ssh_authorized_key` | |
| `ssh_key` | |

## Manage systems with Puppet agent

In a standard Puppet configuration, each node periodically does configuration runs to revert unwanted changes and to pick up recent updates.

On *nix nodes, there are three main ways to do this:

| | |
|---|---|
| **Run Puppet agent as a service.** | The easiest method. The Puppet agent daemon does configuration runs at a set interval, which can be configured. |
| **Make a cron job that runs Puppet agent.** | Requires more manual configuration, but a good choice if you want to reduce the number of persistent processes on your systems. |
| **Only run Puppet agent on demand.** | You can also deploy MCollective to run on demand on many nodes. |

Choose whichever one works best for your infrastructure and culture.

### Run Puppet agent as a service

The `puppet agent` command can start a long-lived daemon process that does configuration runs at a set interval.

**Note:** If you are running Puppet agent as a non-root user, use a cron job instead.

1. **Start the service.**

   The best method is with Puppet agent's init script / service configuration. When you install Puppet with packages, included is an init script or service configuration for controlling Puppet agent, usually with the service name `puppet` (for both open source and Puppet Enterprise).

   In open source Puppet, enable the service by running this command:

   ```
   sudo puppet resource service puppet ensure=running enable=true
   ```

   You can also run the `sudo puppet agent` command with no additional options which causes the Puppet agent to start running and daemonize, however you won't have an interface for restarting or stopping it. To stop the daemon, use the process ID from the agent's `pidfile`:

   ```
   sudo kill $(puppet config print pidfile --section agent)
   ```
2. **(Optional) Configure the run interval.**

   The Puppet agent service defaults to doing a configuration run every 30 minutes. You can configure this with the `runinterval` setting in `puppet.conf`:

   ```
   # /etc/puppetlabs/puppet/puppet.conf
   [agent]
     runinterval = 2h
   ```

   If you don't need frequent configuration runs, a longer run interval lets your Puppet master servers handle many more agent nodes.

### Run Puppet agent as a cron job

Run Puppet agent as a cron job when running as a non-root user.

If the `onetime` setting is set to `true`, the Puppet agent command does one configuration run and then quits. If the `daemonize` setting is set to `false`, the command stays in the foreground until the run is finished. If set to `true`, it does the run in the background.

This behavior is good for building a cron job that does configuration runs. You can use the `splay` and `splaylimit` settings to keep the Puppet master from getting overwhelmed, because the system time is probably synchronized across all of your agent nodes.

To set up a cron job, run the `puppet resource` command:

```
sudo puppet resource cron puppet-agent ensure=present user=root minute=30
 command='/opt/puppetlabs/bin/puppet agent --onetime --no-daemonize --splay
 --splaylimit 60'
```

The above example runs Puppet one time every hour.

### Run Puppet agent on demand

Some sites prefer to run Puppet agent on-demand, and others use scheduled runs along with the occasional on-demand run.

You can start Puppet agent runs while logged in to the target system, or remotely with Bolt or MCollective.

Run Puppet agent on one machine, using SSH to log into it:

```
ssh ops@magpie.example.com sudo puppet agent --test
```

To run remotely on multiple machines, you need some form of orchestration or parallel execution tool, such as Bolt or MCollective with the puppet agent plugin.

**Note:** As of Puppet agent 5.5.4, MCollective is deprecated and will be removed in a future version of Puppet agent. If you use Puppet Enterprise, consider migrating from MCollective to Puppetorchestrator. If you use open source Puppet, migrate MCollective agents and filters using tools like Bolt and PuppetDB's Puppet Query Language.

## Disable and re-enable Puppet runs

Whether you're troubleshooting errors, working in a maintenance window, or developing in a sandbox environment, you might need to temporarily disable the Puppet agent from running.

1. To disable the agent, run:

```
sudo puppet agent --disable "<MESSAGE>"
```

2. To enable the agent, run:

```
sudo puppet agent --enable
```

## Configuring Puppet agent

The Puppet agent comes with a default configuration that you might want to change.

Configure Puppet agent with puppet.conf using the `[agent]` section, the `[main]` section, or both. For information on settings relevant to Puppet agent, see important settings.

### Logging for Puppet agent on *nix systems

When running as a service, Puppet agent logs messages to syslog. Your syslog configuration determines where these messages are saved, but the default location is `/var/log/messages` on Linux, `/var/log/system.log` on Mac OS X, and `/var/adm/messages` on Solaris.

You can adjust how verbose the logs are with the `log_level` setting, which defaults to `notice`.

When running in the foreground with the `--verbose`, `--debug`, or `--test` options, Puppet agent logs directly to the terminal instead of to syslog.

When started with the `--logdest <FILE>` option, Puppet agent logs to the file specified by `<FILE>`.

### Reporting for Puppet agent on *nix systems

In addition to local logging, Puppet agent submits a report to the Puppet master after each run. This can be disabled by setting `report = false` in puppet.conf.)

# Puppet agent on Windows

Puppet agent is the application that manages configurations on your nodes. It requires a Puppet master server to fetch configuration catalogs from.

For more information about invoking the Puppet agent command, see the puppet agent man page.

## Puppet agent's run environment

Puppet agent runs as a specific user, by default `LocalSystem`, and initiates outbound connections on port 8140.

### Ports

By default,Puppet's HTTPS traffic uses port 8140. Your operating system and firewall must allow Puppet agent to initiate outbound connections on this port.

If you want to use a non-default port, change the `masterport` setting on all agent nodes, and ensure that you change your Puppet master's port as well.

#### User

Puppet agent runs as the `LocalSystem` user, which lets it manage the configuration of the entire system, but prevents it from accessing files on UNC shares.

Puppet agent can also run as a different user. You can change the user in the Service Control Manager (SCM). To start the SCM, click **Start** -> **Run…** and then enter `Services.msc`.

You can also specify a different user when installing Puppet. To do this, install using the CLI and specify the required MSI properties: `PUPPET_AGENT_ACCOUNT_USER`, `PUPPET_AGENT_ACCOUNT_PASSWORD`, and `PUPPET_AGENT_ACCOUNT_DOMAIN`.

Puppet agent's user can be a local or domain user. If this user isn't already a local administrator, the Puppet installer adds it to the `Administrators` group. The installer also grants Logon as Service to the user.

## Managing systems with Puppet agent

In a normal Puppet configuration, every node periodically does configuration runs to revert unwanted changes and to pick up recent updates.

On Windows nodes, there are two main ways to do this:

| | |
|---|---|
| **Run Puppet as a service.** | The easiest method. The Puppet agent service does configuration runs at a set interval, which can be configured. |
| **Run Puppet agent on demand.** | You can also use Bolt or deploy MCollective to run on demand on many nodes. |

Because the Windows version of the Puppet agent service is much simpler than the *nix version, there's no real performance to be gained by running Puppet as a scheduled task. If you want scheduled configuration runs, use the Windows service.

### Running Puppet agent as a service

The Puppet installer configures Puppet agent to run as a Windows service and starts it. No further action is needed. Puppet agent does configuration runs at a set interval.

#### Configuring the run interval

The Puppet agent service defaults to doing a configuration run every 30 minutes. If you don't need frequent configuration runs, a longer run interval lets your Puppet master servers handle many more agent nodes.

You can configure this with the `runinterval` setting in `puppet.conf`:

```
# C:\ProgramData\PuppetLabs\puppet\etc\puppet.conf
[agent]
  runinterval = 2h
```

After you change the run interval, the next run happens on the previous schedule, and subsequent runs happen on the new schedule.

#### Configuring the service start up type

The Puppet agent service defaults to starting automatically. If you want to start it manually or disable it, you can configure this during installation.

To do this, install using the CLI and specify the `PUPPET_AGENT_STARTUP_MODE` MSI property.

You can also configure this after installation with the Service Control Manager (SCM). To start the SCM, click **Start** -> **Run...** and enter `Services.msc`.

You can also configure agent service with the `sc.exe` command. To prevent the service from starting on boot, run the following command from the Command Prompt (`cmd.exe`):

```
sc config puppet start= demand
```

**Important:** The space after `start=` is mandatory and must be run in cmd.exe. This command won't work from PowerShell.

To stop and restart the service, run the following commands:

```
sc stop puppet
sc start puppet
```

To change the arguments used when triggering a Puppet agent run, add flags to the command:

```
sc start puppet --debug --logdest eventlog
```

This example changes the level of detail that gets written to the Event Log.

### Running Puppet agent on demand

Some sites prefer to run Puppet agent on demand, and others occasionally need to do an on-demand run.

You can start Puppet agent runs while logged in to the target system, or remotely with Bolt or MCollective.

#### While logged in to the target system

On Windows, log in as an administrator, and start the configuration run by selecting **Start** -> **Run Puppet Agent**. If Windows prompts for User Account Control confirmation, click **Yes**. The status result of the run is shown in a command prompt window.

#### Running other Puppet commands

To run other Puppet-related commands, start a command prompt with administrative privileges. You can do so by right-clicking the **Command Prompt** or **Start Command Prompts with Puppet** program and clicking **Run as administrator**. Click Yes if the system asks for UAC confirmation.

#### Remotely

Open source Puppet users can use Bolt to run tasks and commands on remote systems.

Alternatively, you can install MCollective and the puppet agent plugin to get similar capabilities, but Puppet doesn't provide standalone MCollective packages for Windows.

**Important:** As of Puppet agent 5.5.4, MCollective is deprecated and will be removed in a future version of Puppet agent. If you use Puppet Enterprise, consider migrating from MCollective to Puppet orchestrator. If you use open source Puppet, migrate MCollective agents and filters using tools like Bolt and the PuppetDB Puppet Query Language.

## Disabling and re-enabling Puppet runs

Whether you're troubleshooting errors, working in a maintenance window, or developing in a sandbox environment, you might need to temporarily disable the Puppet agent from running.

1. Start a command prompt with **Run as administrator**.
2. To disable the agent, run:

   ```
   puppet agent --disable "<MESSAGE>"
   ```

3. To enable the agent, run:

```
puppet agent --enable
```

## Configuring Puppet agent on Windows

The Puppet agent comes with a default configuration that you might want to change.

Configure Puppet agent with puppet.conf, using the `[agent]` section, the `[main]` section, or both. For more information on which settings are relevant to Puppet agent, see important settings.

### Logging for Puppet agent on Windows systems

When running as a service, Puppet agent logs messages to the Windows Event Log. You can view its logs by browsing the **Event Viewer**. Click **Control Panel** -> **System and Security** -> **Administrative Tools** -> **Event Viewer**.

By default, Puppet logs to the `Application` event log. However, you can configure Puppet to log to a separate Puppet log instead.

To enable the Puppet log, create the requisite registry key by opening a command prompt and running:

```
reg add HKLM\System\CurrentControlSet\Services\EventLog\Puppet\Puppet /v
 EventMessageFile /t REG_EXPAND_SZ /d "C:\Program Files\Puppet Labs\Puppet
\puppet\bin\puppetres.dll"
```

Alternatively, use PowerShell and the `New-EventLog` cmdlet to do the same thing:

```
New-EventLog -Source Puppet -LogName Puppet -MessageResource "C:\Program
 Files\Puppet Labs\Puppet\puppet\bin\puppetres.dll"
```

**Note:** If you are using an older version of Puppet, double check that you have the most up to date path to `puppetres.dll`.

For existing agents, these commands can be placed in an exec resource to configure agents going forward.

**Note:** Any previously recorded event log messages are not moved; only new messages are recorded in the newly created Puppet log.

You can adjust how verbose the logs are with the `log_level` setting, which defaults to `notice`.

When running in the foreground with the `--verbose`, `--debug`, or `--test` options, Puppet agent logs directly to the terminal.

When started with the `--logdest <FILE>` option, Puppet agent logs to the file specified by `<FILE>`.

### Reporting for Puppet agent on Windows systems

In addition to local logging, Puppet agent submits a report to the Puppet master after each run. This can be disabled by setting `report = false` in puppet.conf.

### Setting Puppet agent CPU priority

When CPU usage is high, lower the priority of the Puppet agent service by using the process priority setting, a cross platform configuration option. Process priority can also be set in the Puppet master configuration.

# Puppet apply

Puppet apply is an application that compiles and manages configurations on nodes. It acts like a self-contained combination of the Puppet master and Puppet agent applications.

For details about invoking the `puppet apply` command, see the puppet apply man page.

### Supported platforms

Puppet apply runs similarly on *nix and Windows systems. Not all operating systems can manage the same resources with Puppet; some resource types are OS-specific, and others have OS-specific features. For more information, see the resource type reference.

## Puppet apply's run environment

Unlike Puppet agent, Puppet apply never runs as a daemon or service. It runs as a single task in the foreground, which compiles a catalog, applies it, files a report, and exits.

By default, it never initiates outbound network connections, although it can be configured to do so, and it never accepts inbound network connections.

### Main manifest

Like the Puppet master application, Puppet apply uses its settings (such as `basemodulepath`) and the configured environments to locate the Puppet code and configuration data it uses when compiling a catalog.

The one exception is the main manifest. Puppet apply always requires a single command line argument, which acts as its main manifest. It ignores the main manifest from its environment.

Alternatively, you can write a main manifest directly using the command line, with the `-e` option. For more information, see the puppet apply man page.

### User

Puppet apply runs as whichever user executed the Puppet apply command.

To manage a complete system, you should run Puppet apply as:

- `root` on *nix systems.
- Either `LocalService` or a member of the `Administrators` group on Windows systems.

Puppet apply can also run as a non-root user. When running without root permissions, most of Puppet's resource providers cannot use `sudo` to elevate permissions. This means Puppet can only manage resources that its user can modify without using `sudo`.

Of the core resource types listed in the resource type reference, the following are available to non-root agents:

| Resource type | Details |
|---|---|
| `augeas` | |
| `cron` | Only non-root cron jobs can be viewed or set. |
| `exec` | Cannot run as another user or group. |
| `file` | Only if the non-root user has read/write privileges. |
| `notify` | |
| `schedule` | |

| Resource type | Details |
|---|---|
| `service` | For services that don't require root. You can also use the `start`, `stop`, and `status` attributes to specify how non-root users should control the service. For more information, see tips and examples for the `service` type. |
| `ssh_authorized_key` | |
| `ssh_key` | |

To install packages into a directory controlled by a non-root user, you can either use an `exec` to unzip a tarball or use a recursive `file` resource to copy a directory into place.

#### Network access

By default, Puppet apply does not communicate over the network. It uses its local collection of modules for any file sources, and does not submit reports to a central server.

Depending on your system and the resources you are managing, it might download packages from your configured package repositories or access files on UNC shares.

If you have configured an external node classifier (ENC), your ENC script might create an outbound HTTP connection. Additionally, if you've configured the HTTP report processor, Puppet agent sends reports via HTTP or HTTPS.

If you have configured PuppetDB, Puppet apply creates outbound HTTPS connections to PuppetDB.

#### Logging

Puppet apply logs directly to the terminal, which is good for interactive use, but less so when running as a scheduled task or cron job.

You can adjust how verbose the logs are with the `log_level` setting, which defaults to `notice`. Setting it to `info` is equivalent to running with the `--verbose` option, and setting it to `debug` is equivalent to `--debug`. You can also make logs quieter by setting it to `warning` or lower.

When started with the `--logdest syslog` option, Puppet apply logs to the *nix syslog service. Your syslog configuration dictates where these messages are saved, but the default location is `/var/log/messages` on Linux, `/var/log/system.log` on Mac OS X, and `/var/adm/messages` on Solaris.

When started with the `--logdest eventlog` option, it logs to the Windows Event Log. You can view its logs by browsing the **Event Viewer**. Click **Control Panel** -> **System and Security** -> **Administrative Tools** -> **Event Viewer**.

When started with the `--logdest <FILE>` option, it logs to the file specified by `<FILE>`.

#### Reporting

In addition to local logging, Puppet apply processes a report using its configured report handlers, like a Puppet master does. Using the `reports` setting, you can enable different reports. For more information, see see the list of available reports. For information about reporting, see the reporting documentation.

To disable reporting and avoid taking up disk space with the `store` report handler, you can set `report = false` in puppet.conf.

### Managing systems with Puppet apply

In a typical site, every node periodically does a Puppet run, to revert unwanted changes and to pick up recent updates.

Puppet apply doesn't run as a service, so you must manually create a scheduled task or cron job if you want it to run on a regular basis, instead of using Puppet agent.

On *nix, you can use the `puppet resource` command to set up a cron job.

This example runs Puppet one time per hour, with Puppet Enterprise paths:

```
sudo puppet resource cron puppet-apply ensure=present user=root minute=60
 command='/opt/puppetlabs/bin/puppet apply /etc/puppetlabs/puppet/manifests
 --logdest syslog'
```

### Configuring Puppet apply

Configure Puppet apply in the `puppet.conf` file, using the `[user]` section, the `[main]` section, or both.

For information on which settings are relevant to `puppet apply`, see important settings.

## Puppet device

With Puppet device, you can manage network devices, such as routers, switches, firewalls, and Internet of Things (IOT) devices, without installing a Puppet agent on them. Devices that cannot run Puppet applications require a Puppet agent to act as a proxy. The proxy manages certificates, collects facts, retrieves and applies catalogs, and stores reports on behalf of a device.

Puppet device runs on both *nix and Windows. The Puppet device application combines some of the functionality of the Puppet apply and Puppet resource applications. For details about running the Puppet device application, see the `puppet device` man page.

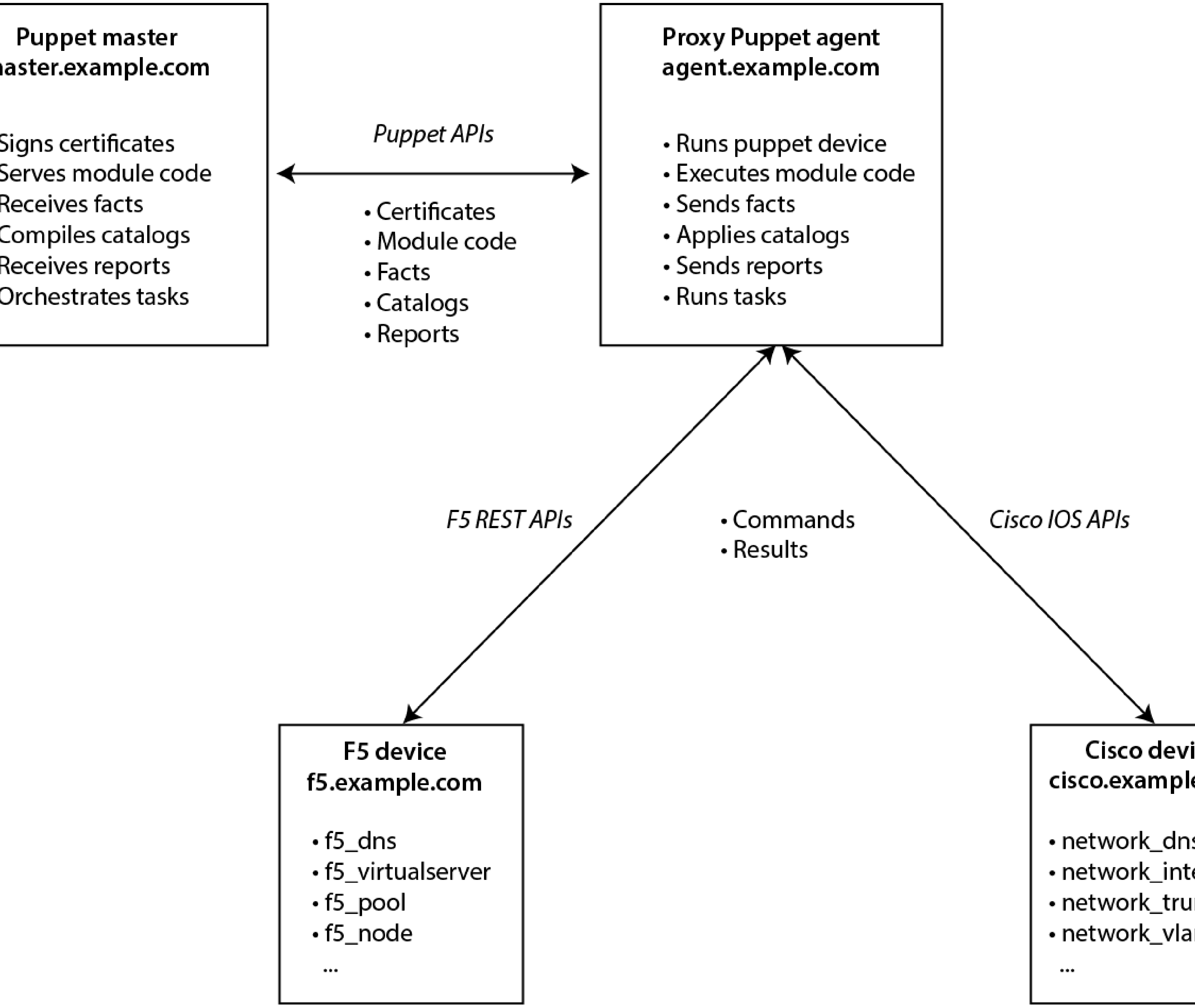
**Note:** If you are writing a module for a remote resource, we recommend using transports instead of devices. Transports have extended functionality and can be used with other workflows, such as with Bolt . For more information on transports and how to port your existing code, see Resource API Transports.

### The Puppet device model

In a typical deployment model, a Puppet agent is installed on each system managed by Puppet. However, not all systems can have agents installed on them.

For these devices, you can configure a Puppet agent on another system which connects to the API or CLI of the device, and acts as a proxy between the device and the Puppet master.

In the diagram below, Puppet device is on a proxy Puppet agent (agent.example.com) and is being used to manage an F5 load balancer (f5.example.com) and a Cisco switch (cisco.example.com).


Puppet master
aster.example.com
Signs certificates
Serves module code
Receives facts
Compiles catalogs
Receives reports
Orchestrates tasks
Puppet APIs
• Certificates
• Module code
• Facts
• Catalogs
• Reports
Proxy Puppet agent
agent.example.com
• Runs puppet device
• Executes module code
• Sends facts
• Applies catalogs
• Sends reports
• Runs tasks
F5 REST APIs
• Commands
• Results
Cisco IOS APIs
F5 device
f5.example.com
• f5_dns
• f5_virtualserver
• f5_pool
• f5_node
...
Cisco devi
cisco.example
• network_dns
• network_inte
• network_tru
• network_vla
...

## Puppet device's run environment

Puppet device runs as a single process in the foreground that manages devices, rather than as a daemon or service like a Puppet agent.

### User

The `puppet device` command runs with the privileges of the user who runs it.

Run Puppet device as:

- Root on *nix
- Either LocalService or a member of the Administrators group on Windows

### Logging

By default, Puppet device outputs directly to the terminal, which is valuable for interactive use. When you run it as a cron job or scheduled task, use the `logdest` option to direct the output to a file.

On *nix, run Puppet device with the `--logdest syslog` option to log to the *nix syslog service:

```
puppet device --verbose --logdest syslog
```

Your syslog configuration determines where these messages are saved, but the default location is `/var/log/messages` on Linux, `/var/log/system.log` on Mac OS X, and `/var/adm/messages` on Solaris. For example, to view these logs on Linux, run:

```
tail /var/log/messages
```

On Windows, run Puppet device with the `--logdest eventlog` option, which logs to the Windows Event Log, for example:

```
puppet device --verbose --logdest eventlog
```

To view these logs on Windows, click **Control Panel** → **System and Security** → **Administrative Tools** → **Event Viewer**.

To specify a particular file to send Puppet device log messages to, use the `--logdest <FILE> option`, which logs to the file specified by `<FILE>`, for example:

```
puppet device --verbose --logdest /var/log/puppetlabs/puppet/device.log
```

You can increase the logging level with the `--debug` and `--verbose` options.

In addition to local logging, Puppet device submits reports to the Puppet master after each run. These reports contain standard data from the Puppet run, including any corrective changes.

### Network access

Puppet device creates outbound network connections to the devices it manages. It requires network connectivity to the devices via their API or CLI. It never accepts inbound network connections.

## Installing device modules

You need to install the device module for each device you want to manage on the Puppet master.

For example, to install the f5 and cisco_ios device modules on the Puppet master, run the following commands:

```
$ sudo puppet module install f5-f5
```

```
$ sudo puppet module install puppetlabs-cisco_ios
```

## Configuring Puppet device on the proxy Puppet agent

You can specify multiple devices in `device.conf`, which is configurable with the `deviceconfig` setting on the proxy agent.

For example, to configure an F5 and a Cisco IOS device, add the following lines to the `device.conf` file:

```
[f5.example.com]
type f5
url https://username:password@f5.example.com

[cisco.example.com]
type cisco_ios
url file:///etc/puppetlabs/puppet/devices/cisco.example.com.yaml
```

The string in the square brackets is the device's certificate name — usually the hostname or FQDN. The certificate name is how Puppet identifies the device.

For the `url`, specify the device's connection string. The connection string varies by device module. In the first example above, the F5 device connection credentials are included in the `url` `device.conf` file, because that is how the F5 module stores credentials. However, the Cisco IOS module uses the Puppet Resource API, which stores that information in a separate credentials file. So, Cisco IOS devices would also have a `/etc/puppetlabs/puppet/devices/<device cert name>.conf` file similar to the following content:

```
{
"address": "cisco.example.com"
"port": 22
"username": "username"
"password": "password"
"enable_password": "password"
}
}
```

For more information, see `device.conf`.

## Classify the proxy Puppet agent for the device

Some device modules require the proxy Puppet agent to be classified with the base class of the device module to install or configure resources required by the module. Refer to the specific device module README for details.

To classify proxy Puppet agent:

1. Classify the agent with the base class of the device module, for each device it manages in the manifest. For example:

   ```
   node 'agent.example.com' {
     include cisco_ios
     include f5
   }
   ```
2. Apply the classification by running `puppet agent -t` on the proxy Puppet agent.

## Classify the device

Classify the device with resources to manage its configuration.

The examples below manage DNS settings on an F5 and a Cisco IOS device.

1. In the `site.pp` manifest, declare DNS resources for the devices. For example:

   ```
   node 'f5.example.com' {
    f5_dns{ '/Common/dns':
     name_servers => ['4.2.2.2.', '8.8.8.8"],
     same     => ['localhost",' example.com'],
   ```

```
 }
}

node 'cisco.example.com' {
 network_dns { 'default':
  servers => [4.2.2.2', '8.8.8.8'],
  search => ['localhost",'example.com'],
 }
}
```

2. Apply the manifest by running `puppet device -v` on the proxy Puppet agent.

**Note:** Resources vary by device module. Refer to the specific device module README for details.

## Get and set data using Puppet device

The traditional Puppet apply and Puppet resource applications cannot target device resources: running `puppet resource --target <DEVICE>` does not return data from the target device. Instead, use Puppet device to get data from devices, and to set data on devices. The following are optional parameters.

### Get device data with the `resource` parameter

Syntax:

```
puppet device --resource <RESOURCE> --target <DEVICE>
```

Use the `resource` parameter to retrieve resources from the target device. For example, to return the DNS values for example F5 and Cisco IOS devices:

```
sudo puppet device --resource f5_dns --target f5.example.com
sudo puppet device --resource network_dns --target cisco.example.com
```

### Set device data with the `apply` parameter

Syntax:

```
puppet device --verbose --apply <FILE> --target <DEVICE>
```

Use the `--apply` parameter to set a local manifest to manage resources on a remote device. For example, to apply a Puppet manifest to the F5 and Cisco devices:

```
sudo puppet device --verbose --apply manifest.pp --target f5.example.com
sudo puppet device --verbose --apply manifest.pp --target cisco.example.com
```

### View device facts with the `facts` parameter

Syntax:

```
puppet device --verbose --facts --target <DEVICE>
```

Use the `--facts` parameter to display the facts of a remote target. For example, to display facts on a device:

```
sudo puppet device --verbose --facts --target f5.example.com
```

## Managing devices using Puppet device

Running the `puppet device` or `puppet-device` command (without `--resource` or `--apply` options) tells the proxy agent to retrieve catalogs from the master and apply them to the remote devices listed in the `device.conf` file.

To run Puppet device on demand and for all of the devices in `device.conf` , run:

```
sudo puppet device --verbose
```

To run Puppet device for only one of the multiple devices in the `device.conf` file, specify a `--target` option:

```
$ sudo puppet device -verbose --target f5.example.com
```

To run Puppet device on a specific group of devices, as opposed to all devices in the `device.conf` file, create a separate configuration file containing the devices you want to manage, and specify the file with the `--deviceconfig` option:

```
$ sudo puppet device --verbose --deviceconfig /path/to/custom-device.conf
```

To set up a cron job to run Puppet device on a recurring schedule, run:

```
$ sudo puppet resource cron puppet-device ensure=present user=root minute=30
 command='/opt/puppetlabs/bin/puppet device --verbose --logdest syslog'
```

### Example

Follow the steps below to run Puppet device in a production environment, using `cisco_ios` as an example.

1. Install the module on the Puppet master: `sudo puppet module install puppetlabs-cisco_ios`.
2. Include the module on the proxy Puppet agent by adding the following line to the master's `site.pp` file:

   ```
   include cisco_ios
   ```
3. Edit `device.conf` on the proxy Puppet agent:

   ```
   [cisco.example.com]
   type cisco_ios
   url file:///etc/puppetlabs/puppet/devices/cisco.example.com.yaml
   ```
4. Create the `cisco.example.com` credentials file required by modules that use the Puppet Resource API:

   ```
   {
     "address": "cisco.example.com"
     "port": 22
     "username": "username"
     "password": "password"
     "enable_password": "password"
   }
   ```
5. Request a certificate on the proxy Puppet agent: `sudo puppet device --verbose --waitforcert 0 --target cisco.example.com`
6. Sign the certificate on the master: `sudo puppet cert sign cisco.example.com`
7. Run `puppet device` on the proxy Puppet agent to test the credentials: `sudo puppet device --target cisco.example.com`

### Automating device management using the puppetlabs device_manager module

The `puppetlabs-device_manager` module manages the configuration files used by the Puppet device application, applies the base class of configured device modules, and provides additional resources for scheduling and orchestrating Puppet device runs on proxy Puppet agents.

For more information, see the module README.

### Troubleshooting Puppet device

These options are useful for troubleshooting Puppet device command results.

| | |
|---|---|
| `--debug` or `-d` | Enables debugging |
| `--trace` or `-t` | Enables stack tracing if Ruby fails |
| `--verbose` or `-v` | Enables detailed reporting |

# Puppet Server

## Using and extending Puppet Server

## Known issues and workarounds

## Administrative API endpoints

## CA v1 API

## Server-specific Puppet API endpoints

## Status API endpoints

## Metrics API endpoints

## Developer information

# The Puppet language

You'll use Puppet's declarative language to describe the desired state of your system in files called manifests. Manifests describe how your network and operating system resources, such as files, packages, and services, should be configured. Puppet then compiles those manifests into catalogs, and applies each catalog to its corresponding node to ensure the node is configured correctly, across your infrastructure.

Several parts of the Puppet language depend on evaluation order. For example, variables must be set before they are referenced. Throughout the language reference, we call out areas where the order of statements matters.

• Language visual index on page 173
A quick reference of syntax examples for the Puppet language.
• The Puppet language style guide on page 178
This style guide promotes consistent formatting in the Puppet language, giving you a common pattern, design, and style to follow when developing modules. This consistency in code and module structure makes it easier to update and maintain the code.
• Files and paths on Windows on page 202
Puppet and Windows handle directory separators and line endings in files somewhat differently, so you must be aware of the differences when you are writing manifests to manage Windows systems.
• Code comments on page 203
To add comments to your Puppet code, use shell-style or hash comments.
• Variables on page 204
Variables store values so that those values can be accessed in code later.
• Resources on page 207
Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.
• Relationships and ordering on page 214
Puppet normally applies resources in the order they are declared in their manifest. To manage a group of resources in a specific order, explicitly declare such relationships with relationship metaparameters, chaining arrows, and the `require` function.
• Classes on page 220
*Classes* are named blocks of Puppet code that are stored in modules and applied later when they are invoked by name. You can add classes to a node's catalog by either declaring them in your manifests or assigning them from an external node classifier (ENC). Classes generally configure large or medium-sized chunks of functionality, such as all of the packages, configuration files, and services needed to run an application.
• Defined resource types on page 227
*Defined resource types*, sometimes called defined types or defines, are blocks of Puppet code that can be evaluated multiple times with different parameters.
• Bolt tasks on page 230
Bolt tasks are single actions that you can run on target nodes in your infrastructure, allowing you to make as-needed changes to remote systems. You can run tasks with the Puppet Enterprise (PE) orchestrator or with Puppet's standalone task runner, Bolt.
• Type aliases on page 231
Type aliases allow you to create reusable and descriptive data types and resource types.
• Expressions and operators on page 232
*Expressions* are statements that resolve to values. You can use expressions almost anywhere a value is required. Expressions can be compounded with other expressions, and the entire combined expression resolves to a single value.
• Conditional statements and expressions on page 242
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. Puppet supports *if* and *unless* statements, *case* statements, and *selectors*.
• Function calls on page 249
Functions are plug-ins, written in Ruby, that you can call during catalog compilation. A call to any function is an expression that resolves to a value. Most functions accept one or more values as arguments, and return a resulting value.
• Built-in functions

- Node definitions on page 252

A node definition, also known as a node statement, is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.
- Facts and built-in variables on page 254

Before requesting a catalog for a managed node, or compiling one with `puppet apply`, Puppet collects system information, called *facts*, by using the Facter tool. The facts are assigned as values to variables that you can use anywhere in your manifests. Puppet also sets some additional special variables, called *built-in variables*, which behave a lot like facts.
- Reserved words and acceptable names on page 260

You can use only certain characters for naming variables, modules, classes, defined types, and other custom constructs. Additionally, some words in the Puppet language are reserved and cannot be used as bare word strings or names.
- Custom resources on page 265

A *resource* is the basic unit that is managed by Puppet. Each resource has a set of attributes describing its state. Some attributes can be changed throughout the lifetime of the resource, whereas others are only reported back but cannot be changed, and some can only be set one time during initial creation.
- Values and data types on page 294

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.
- Templates on page 336

Templates are documents that combine code, data, and literal text to produce a final rendered output. The goal of a template is to manage a complicated piece of text with simple inputs.
- Advanced constructs on page 352

Advanced Puppet language constructs help you write simpler and more effective Puppet code by reducing complexity.
- Details of complex behaviors on page 367

Within Puppet language there are complex behavior patterns regarding classes, defined types, and specific areas of code called scopes.

#### Related information

Puppet language and code on page 14
You'll use Puppet's declarative language to describe the desired state of your system in files called manifests. Manifests describe how your network and operating system resources, such as files, packages, and services, should be configured. Puppet then compiles those manifests into catalogs, which across your infrastructure.

Language visual index on page 173
A quick reference of syntax examples for the Puppet language.

The Puppet language style guide on page 178
This style guide promotes consistent formatting in the Puppet language, giving you a common pattern, design, and style to follow when developing modules. This consistency in code and module structure makes it easier to update and maintain the code.

## Language visual index

A quick reference of syntax examples for the Puppet language.

### Resource declaration

This example resource declaration includes:

- `file`: The *resource type*.
- `ntp.conf`: The resource *title*.
- `path`: An *attribute*.
- `'/etc/ntp.conf'`: The value of an attribute; 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.

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

For details about resources and resource declaration syntax, see Resources on page 207.

#### Resource relationship metaparameters

Two resource declarations establishing relationships with the `before` and `subscribe` *metaparameters*, which accept resource references.

The first declaration ensures that the `ntp` package is installed before the `ntp.conf` file is created. The second declaration ensures that the `ntpd` service is notified of any changes to the `ntp.conf` file.

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

For details about relationships usage and syntax, see Relationships and ordering on page 214. For details about resource references, see Resource and class references on page 320.

#### Resource relationship chaining arrows

Chaining arrows forming relationships between three resources, using resource references.In this example, the `ntp` package must be installed before the `ntp.conf` file is created; after the file is created, the `ntpd` service is notified.

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

For details about relationships usage and syntax, see Relationships and ordering on page 214. For details about resource references, see Resource and class references on page 320.

#### Variable assigned an array value

A variable being assigned a set of values as an array.

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

For details about assigning values to variables, see Variables on page 204.

#### Variable assigned a hash value

A variable being assigned a set of values as a hash.

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

For details about assigning values to variables, see Variables on page 204.

### Interpolated variable

A built-in variable provided by the master being interpolated into a double-quoted string.

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

For details about built-in variables usage and syntax, see Facts and built-in variables on page 254. For information about strings and interpolation, see Strings on page 295.

### Class definition

A class definition, which makes a class available for later use.

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

For details about class usage and syntax, see Classes on page 220.

### Class declarations

Declaring the `ntp` class in three different ways:

- the `include` function
- the `require` function
- the resource-like syntax

Declaring a class causes the resources in it to be managed.

The `include` function is the standard way to declare classes:

```
include ntp
```

The `require` function declares the class and makes it a dependency of the code container where it is declared:

```
require ntp
```

The resource-like syntax declares the class and applies resource-like behavior. Resource-like class declarations require that you declare a given class only one time.

```
class {'ntp':}
```

For details about class usage and syntax, see Classes on page 220.

### Defined resource type definition

Defining a type creates a new defined resource type. The name of this *defined type* has two namespace segments, comprising the name of the module containing the defined type, `apache`, and the name of the defined type itself, `vhost`.

```
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'],
  }
```

For details about defined type usage and syntax, see Defined resource types on page 227.

### Defined type resource declaration

Declarations of an instance, or resource, of a defined type are similar to other resource declarations. This example declares a instance of the `apache::vhost` defined type, with a title of "homepages" and the `port` and `docroot` attributes specified.

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

For details about defined type usage and syntax, see Defined resource types on page 227.

### Defined type resource reference

A resource reference to an instance of the `apache::vhost` defined resource. Every namespace segment in a resource reference must be capitalized.

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

For details about defined type usage and syntax, see Defined resource types on page 227. For details about resource references, see Resource and class references on page 320.

### Node definition

A node definition or node statement is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.

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

Node names in node definitions can also be given as regular expressions.

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

For details about node definition usage and syntax, see Node definitions on page 252.

### `if` statement, using expressions and facts

An `if` statement, whose conditions are expressions that use facts provided by the agent.

```
if $is_virtual {
  warning( 'Tried to include class ntp on virtual machine; this node might
 be misclassified.' )
}
elsif $operatingsystem == 'Darwin' {
```

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

For details about `if` statements, see Conditional statements and expressions on page 242.

### `if` statement, with `in` expression

An `if` statement using an `in` expression.

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

For details about `if` statements, see Conditional statements and expressions on page 242.

### Case statement

A case statement.

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

For details about case statements, see Conditional statements and expressions on page 242.

### Selector statement

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

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

For details about selector statements, see Conditional statements and expressions on page 242.

### Resource default for the exec type

A resource default statement set default attribute values for a given resource type. This example specifies defaults for the `exec` resource type attributes `path`, `environment`, `logoutput`, and `timeout`.

```
Exec {
  path        => '/usr/bin:/bin:/usr/sbin:/sbin',
  environment => 'RUBYLIB=/opt/puppetlabs/puppet/lib/ruby/site_ruby/2.1.0/',
  logoutput   => true,
  timeout     => 180,
}
```

For details about default statement usage and syntax, see Resource defaults.

### Resource collector

A *resource collector*, sometimes called the "spaceship operator." Resource collectors select a group of resources by searching the attributes of each resource in the catalog.

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

For details about resource collector usage and syntax, see Resource collectors.

### Exported resource collector

An exported resource collector, which works with *exported resources*, which are available for use by other nodes.

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

For details about resource collector usage and syntax, see Resource collectors. For information about declaring and collecting exported resources, see Exported resources.

### Exported resource declaration

An *exported 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],
}
```

For information about declaring and collecting exported resources, see Exported resources.

### Virtual resource

A *virtual resource*, which is declared in the catalog but isn't applied to a system unless it is explicitly *realized*.

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

For details about virtual resource usage and syntax, see Virtual resources.

## The Puppet language style guide

This style guide promotes consistent formatting in the Puppet language, giving you a common pattern, design, and style to follow when developing modules. This consistency in code and module structure makes it easier to update and maintain the code.

This style guide applies to Puppet 4 and later. Puppet 3 is no longer supported, but we include some Puppet 3 guidelines in case you're maintaining older code.

**Tip:** Use puppet-lint and metadata-json-lint to check your module for compliance with the style guide.

No style guide can cover every circumstance you might run into when developing Puppet code. When you need to make a judgement call, keep in mind a few general principles.

| | |
|---|---|
| **Readability matters** | If you have to choose between two equal alternatives, pick the more readable one. This is subjective, but if you can read your own code three months from now, it's a great start. In particular, code that generates readable diffs is highly preferred. |
| **Scoping and simplicity are key** | When in doubt, err on the side of simplicity. A module should contain related resources that enable it to accomplish a task. If you describe the function of your module and you find yourself using the word "and," consider splitting the module. Have one goal, with all your classes and parameters focused on achieving it. |
| **Your module is a piece of software** | At least, we recommend that you treat it that way. When it comes to making decisions, choose the option that is easier to maintain in the long term. |

These guidelines apply to Puppet code, for example, code in Puppet modules or classes. To reduce repetitive phrasing, we don't include the word 'Puppet' in every description, but you can assume it.

For information about the specific meaning of terms like 'must,' 'must not,' 'required,' 'should,' 'should not,' 'recommend,' 'may,' and 'optional,' see RFC 2119.

## Module design practices

Consistent module design practices makes module contributions easier.

### Spacing, indentation, and whitespace

Module manifests should follow best practices for spacing, indentation, and whitespace.

Manifests:

- Must use two-space soft tabs.
- Must not use literal tab characters.
- Must not contain trailing whitespace.
- Must include trailing commas after all resource attributes and parameter definitions.
- Must end the last line with a new line.
- Must use one space between the resource type and opening brace, one space between the opening brace and the title, and no spaces between the title and colon.

  Good:

  ```
  file { '/tmp/sample':
  ```

  Bad: Space between title and colon:

  ```
  file { '/tmp/sample' :
  ```

  Bad: No spaces:

  ```
  file{'/tmp/sample':
  ```

  Bad: Too many spaces:

  ```
  file     { '/tmp/sample':
  ```
- Should not exceed a 140-character line width, except where such a limit would be impractical.
- Should leave one empty line between resources, except when using dependency chains.

- May align hash rockets (`=>`) within blocks of attributes, one space after the longest resource key, arranging hashes for maximum readability first.

### Arrays and hashes

To increase readability of arrays and hashes, it is almost always beneficial to break up the elements on separate lines.

Use a single line only if that results in overall better readability of the construct where it appears, such as when it is very short. When breaking arrays and hashes, they should have:

- Each element on its on line.
- Each new element line indented one level.
- First and last lines used only for the syntax of that data type.

Good: Array with multiple elements on multiple lines:

```
service { 'sshd':
  require => [
    Package['openssh-server'],
    File['/etc/ssh/sshd_config'],
  ],
}
```

Good: Hash with multiple elements on multiple lines:

```
$myhash = {
  key       => 'some value',
  other_key => 'some other value',
}
```

Bad: Array with multiple elements on same line:

```
service { 'sshd':
  require => [ Package['openssh-server'], File['/etc/ssh/sshd_config'], ],
}
```

Bad: Hash with multiple elements on same line:

```
$myhash = { key => 'some value', other_key => 'some other value', }
```

Bad: Array with multiple elements on different lines, but syntax and element share a line:

```
service { 'sshd':
  require => [ Package['openssh-server'],
    File['/etc/ssh/sshd_config'],
  ],
}
```

Bad: Hash with multiple elements on different lines, but syntax and element share a line:

```
$myhash = { key => 'some value',
  other_key     => 'some other value',
}
```

Bad: Array with idention of elements past two spaces:

```
service { 'sshd':
  require => [
              Package['openssh-server'],
              File['/etc/ssh/sshd_config'],
  ],
```

```
}
```

### Quoting

As long you are consistent, strings may be enclosed in single or double quotes, depending on your preference.

Regardless of your preferred quoting style, all variables MUST be enclosed in braces when interpolated in a string.

For example:

Good:

```
"/etc/${file}.conf"
```

```
"${facts['operatingsystem']} is not supported by ${module_name}"
```

Bad:

```
"/etc/$file.conf"
```

#### Option 1: Prefer single quotes

Modules that adopt this string quoting style MUST enclose all strings in single quotes, except as listed below.

For example:

Good:

```
owner => 'root'
```

Bad:

```
owner => "root"
```

A string MUST be enclosed in double quotes if it:

- Contains variable interpolations.
  - Good:

    ```
    "/etc/${file}.conf"
    ```
  - Bad:

    ```
    '/etc/${file}.conf'
    ```
- Contains escaped characters not supported by single-quoted strings.
  - Good:

    ```
    content => "nameserver 8.8.8.8\n"
    ```
  - Bad:

    ```
    content => 'nameserver 8.8.8.8\n'
    ```

A string SHOULD be enclosed in double quotes if it:

- Contains single quotes.
  - Good:

    ```
    warning("Class['apache'] parameter purge_vdir is deprecated in favor of
     purge_configs")
    ```
  - Bad:

    ```
    warning('Class[\'apache\'] parameter purge_vdir is deprecated in favor
     of purge_configs')
    ```

**Option 2: Prefer double quotes**

Modules that adopt this string quoting style MUST enclose all strings in double quotes, except as listed below.

For example:

Good:

```
owner => "root"
```

Bad:

```
owner => 'root'
```

A string SHOULD be enclosed in single quotes if it does not contain variable interpolations AND it:

- Contains double quotes.
  - Good:

    ```
    warning('Class["apache"] parameter purge_vdir is deprecated in favor of
     purge_configs')
    ```
  - Bad:

    ```
    warning("Class[\"apache\"] parameter purge_vdir is deprecated in favor
     of purge_configs")
    ```
- Contains literal backslash characters that are not intended to be part of an escape sequence.
  - Good:

    ```
    path => 'c:\windows\system32'
    ```
  - Bad:

    ```
    path => "c:\\windows\\system32"
    ```

If a string is a value from an enumerable set of options, such as `present` and `absent`, it SHOULD NOT be enclosed in quotes at all.

For example:

Good:

```
ensure => present
```

Bad:

```
ensure => "present"
```

### Escape characters

Use backslash (`\`) as an escape character.

For both single- and double-quoted strings, escape the backslash to remove this special meaning: `\\` This means that for every backslash you want to include in the resulting string, use two backslashes. As an example, to include two literal backslashes in the string, you would use four backslashes in total.

Do not rely on unrecognized escaped characters as a method for including the backslash and the character following it.

Unicode character escapes using fewer than 4 hex digits, as in `\u040`, results in a backslash followed by the string `u040`. (This also causes a warning for the unrecognized escape.) To use a number of hex digits not equal to 4, use the longer `u{digits}` format.

### Comments

Comments must be hash comments (`# This is a comment`). Comments should explain the why, not the how, of your code.

Do not use `/* */` comments in Puppet code.

Good:

```
# Configures NTP
file { '/etc/ntp.conf': ... }
```

Bad:

```
/* Creates file /etc/ntp.conf */
file { '/etc/ntp.conf': ... }
```

**Note:** Include documentation comments for Puppet Strings for each of your classes, defined types, functions, and resource types and providers. If used, documentation comments precede the name of the element. For documentation recommendations, see the Modules section of this guide.

### Functions

Avoid the `inline_template()` and `inline_epp()` functions for templates of more than one line, because these functions don't permit template validation. Instead, use the `template()` and `epp()` functions to read a template from the module. This method allows for syntax validation.

You should avoid using calls to Hiera functions in modules meant for public consumption, because not all users have implemented Hiera. Instead, we recommend using parameters that can be overridden with Hiera.

### Improving readability when chaining functions

In most cases, especially if blocks are short, we recommend keeping functions on the same line. If you have a particularly long chain of operations or block that you find difficult to read, you can break it up on multiples lines to improve readability. As long as your formatting is consistent throughout the chain, it is up to your own judgment.

For example, this:

```
$foodgroups.fruit.vegetables
```

Is better than this:

```
   $foodgroups
           .fruit
           .vegetables
```

But, this:

```
$foods = {
 "avocado"    => "fruit",
 "eggplant"   => "vegetable",
 "strawberry" => "fruit",
 "raspberry"  => "fruit",
}

$berries = $foods.filter |$name, $kind| {
 # Choose only fruits
 $kind == "fruit"
}.map |$name, $kind| {
 # Return array of capitalized fruits
 String($name, "%c")
}.filter |$fruit| {
 # Only keep fruits named "berry"
 $fruit =~ /berry$/
}
```

Is better than this:

```
$foods = {
 "avocado"    => "fruit",
 "eggplant"   => "vegetable",
 "strawberry" => "fruit",
 "raspberry"  => "fruit",
}

$berries = $foods.filter |$name, $kind| { $kind == "fruit" }.map |$name,
 $kind| { String($name, "%c") }.filter |$fruit| { $fruit =~ /berry$/ }
```

#### Related information

Modules on page 200
Develop your module using consistent code and module structures to make it easier to update and maintain.

## Resources

Resources are the fundamental unit for modeling system configurations. Resource declarations have a lot of possible features, so your code's readability is crucial.

### Resource names

All resource names or titles must be quoted. If you are using an array of titles you must quote each title in the array, but cannot quote the array itself.

Good:

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

Bad:

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

These quoting requirements do not apply to expressions that evaluate to strings.

### Arrow alignment

To align hash rockets (=>) in a resource's attribute/value list or in a nested block, place the hash rocket one space ahead of the longest attribute name. Indent the nested block by two spaces, and place each attribute on a separate line. Declare very short or single purpose resource declarations on a single line.

Good:

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

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

myresource { 'test':
  ensure => present,
  myhash => {
    'myhash_key1' => 'value1',
    'key2'        => 'value2',
  },
}

notify { 'warning': message => 'This is an example warning' }
```

Bad:

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

file { "/path/to/my-filename.txt":
  ensure => file, mode => $mode, owner => $owner, group => $group,
  source => 'puppet:///modules/my-module/productions/my-filename.txt'
}
```

### Attribute ordering

If a resource declaration includes an `ensure` attribute, it should be the first attribute specified so that a user can quickly see if the resource is being created or deleted.

Good:

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

When using the special attribute `*` (asterisk or splat character) in addition to other attributes, splat should be ordered last so that it is easy to see. You may not include multiple splats in the same body.

Good:

```
$file_ownership = {
  'owner' => 'root',
  'group' => 'wheel',
  'mode'  => '0644',
}

file { '/etc/passwd':
  ensure => file,
  *      => $file_ownership,
```

```
}
```

### Resource arrangement

Within a manifest, resources should be grouped by logical relationship to each other, rather than by resource type.

Good:

```
file { '/tmp/dir':
  ensure => directory,
}

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

file { '/tmp/dir2':
  ensure => directory,
}

file { '/tmp/dir2/b':
  content => 'b',
}
```

Bad:

```
file { '/tmp/dir':
  ensure => directory,
}

file { '/tmp/dir2':
  ensure => directory,
}

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

file { '/tmp/dir2/b':
  content => 'b',
}
```

Use semicolon-separated multiple resource bodies only in conjunction with a local default body.

Good:

```
$defaults = { < hash of defaults > }

file {
  default:
    * => $defaults,;

  '/tmp/testfile':
    content => 'content of the test file',
}
```

Good: Repeated pattern with defaults:

```
$defaults = { < hash of defaults > }

file {
  default:
```

```
    * => $defaults,;

  '/tmp/motd':
    content => 'message of the day',;

  '/tmp/motd_tomorrow':
    content => 'tomorrows message of the day',;
}
```

Bad: Unrelated resources grouped:

```
file {
  '/tmp/testfile':
    owner    => 'admin',
    mode     => '0644',
    contents => 'this is the content',;

  '/opt/myapp':
    owner  => 'myapp-admin',
    mode   => '0644',
    source => 'puppet://<someurl>',;

  # etc
}
```

You cannot set any attribute more than one time for a given resource; if you try, Puppet raises a compilation error. This means:

- If you use a hash to set attributes for a resource, you cannot set a different, explicit value for any of those attributes. For example, if mode is present in the hash, you can't also set `mode => "0644"` in that resource body.
- You can't use the `*` attribute multiple times in one resource body, because `*` itself acts like an attribute.
- To use some attributes from a hash and override others, either use a hash to set per-expression defaults, or use the `+` (merging) operator to combine attributes from two hashes (with the right-hand hash overriding the left-hand one).

### Symbolic links

Declare symbolic links with an ensure value of `ensure => link`. To inform the user that you are creating a link, specify a value for the `target` attribute.

Good:

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

Bad:

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

### File modes

- POSIX numeric notation must be represented as 4 digits.
- POSIX symbolic notation must be a string.
- You should not use file mode with Windows; instead use the acl module.
- You should use numeric notation whenever possible.

- The file mode attribute should always be a quoted string or (unquoted) variable, never an integer.

Good:

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

Bad:

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

### Multiple resources

Multiple resources declared in a single block should be used only when there is also a default set of options for the resource type.

Good:

```
file {
  default:
    ensure => 'file',
    mode   => '0666',;

  '/owner':
    user => 'owner',;

  '/staff':
    user => 'staff',;
}
```

Good: Give the defaults a name if used several times:

```
$our_default_file_attributes = {
  'ensure' => 'file',
  'mode'   => '0666',
}

file {
  default:
    * => $our_default_file_attributes,;

  '/owner':
    user => 'owner',;

  '/staff':
    user => 'staff',;
}
```

Good: Spell out 'magic' iteration:

```
['/owner', '/staff'].each |$path| {
  file { $path:
    ensure => 'file',
  }
}
```

Good: Spell out 'magic' iteration:

```
$array_of_paths.each |$path| {
  file { $path:
    ensure => 'file',
  }
}
```

Bad:

```
file {
  '/owner':
    ensure => 'file',
    user   => owner,
    mode   => '0666',;

  '/staff':
    ensure => 'file',
    user   => staff,
    mode   => '0774',;
}

file { ['/owner', '/staff']:
  ensure => 'file',
}

file { $array_of_paths:
  ensure => 'file',
}
```

### Legacy style defaults

Avoid legacy style defaults. If you do use them, they should occur only at top scope in your site manifest. This is because resource defaults propagate through dynamic scope, which can have unpredictable effects far away from where the default was declared.

Acceptable: `site.pp`:

```
Package {
  provider => 'zypper',
}
```

Bad: `/etc/puppetlabs/puppet/modules/apache/manifests/init.pp`:

```
File {
  owner => 'nobody',
  group => 'nogroup',
  mode  => '0600',
}

concat { $config_file_path:
  notify  => Class['Apache::Service'],
  require => Package['httpd'],
}
```

### Attribute alignment

Resource attributes must be uniformly indented in two spaces from the title.

Good:

```
file { '/owner':
  ensure => 'file',
  owner  => 'root',
}
```

Bad: Too many levels of indentation:

```
file { '/owner':
    ensure => 'file',
    owner  => 'root',
}
```

Bad: No indentation:

```
file { '/owner':
ensure => 'file',
owner  => 'root',
}
```

Bad: Improper and non-uniform indentation:

```
file { '/owner':
  ensure => 'file',
   owner => 'root',
}
```

Bad: Indented the wrong direction:

```
  file { '/owner':
ensure => 'file',
owner  => 'root',
  }
```

For multiple bodies, each title should be on its own line, and be indented. You may align all arrows across the bodies, but arrow alignment is not required if alignment per body is more readable.

```
file {
  default:
    * => $local_defaults,;

  '/owner':
    ensure => 'file',
    owner  => 'root',
}
```

### Defined resource types

Because defined resource types can have multiple instances, resource names must have a unique variable to avoid duplicate declarations.

Good: Template uses `$listen_addr_port`:

```
define apache::listen {
  $listen_addr_port = $name

  concat::fragment { "Listen ${listen_addr_port}":
    ensure  => present,
    target  => $::apache::ports_file,
```

```
    content => template('apache/listen.erb'),
  }
}
```

Bad: Template uses `$name`:

```
define apache::listen {

  concat::fragment { 'Listen port':
    ensure  => present,
    target  => $::apache::ports_file,
    content => template('apache/listen.erb'),
  }
}
```

## Classes and defined types

Classes and defined types should follow scope and organization guidelines.

### Separate files

Put all classes and resource type definitions (defined types) as separate files in the `manifests` directory of the module. Each file in the manifest directory should contain nothing other than the class or resource type definition.

Good: `etc/puppetlabs/puppet/modules/apache/manifests/init.pp`:

```
class apache { }
```

Good: `etc/puppetlabs/puppet/modules/apache/manifests/ssl.pp`:

```
class apache::ssl { }
```

Good: `etc/puppetlabs/puppet/modules/apache/manifests/virtual_host.pp`:

```
define apache::virtual_host () { }
```

Separating classes and defined types into separate files is functionally identical to declaring them in `init.pp`, but has the benefit of highlighting the structure of the module and making the function and structure more legible.

When a resource or include statement is placed outside of a class, node definition, or defined type, it is included in all catalogs. This can have undesired effects and is not always easy to detect.

Good: `manifests/init.pp`:

```
# class ntp
class ntp {
  ntp::install
}
# end of file
```

Bad: `manifests/init.pp`:

```
class ntp {
  #...
}
ntp::install
```

### Internal organization of classes and defined types

Structure classes and defined types to accomplish one task.

Documentation comments for Puppet Strings should be included for each class or defined type. If used, documentation comments must precede the name of the element. For complete documentation recommendations, see the Modules section.

Put the lines of code in the following order:

1. First line: Name of class or type.
2. Following lines, if applicable: Define parameters. Parameters should be typed.
3. Next lines: Includes and validation come after parameters are defined. Includes may come before or after validation, but should be grouped separately, with all includes and requires in one group and all validations in another. Validations should validate any parameters and fail catalog compilation if any parameters are invalid. Seen ntp for an example
4. Next lines, if applicable: Should declare local variables and perform variable munging.
5. Next lines: Should declare resource defaults.
6. Next lines: Should override resources if necessary.

The following example follows the recommended style.

In `init.pp`:

- The `myservice` class installs packages, ensures the state of `myservice`, and creates a tempfile with given content. If the tempfile contains digits, they are filtered out.
- `@param service_ensure` the wanted state of services.
- `@param package_list` the list of packages to install, at least one must be given, or an error of unsupported OS is raised.
- `@param tempfile_contents` the text to be included in the tempfile, all digits are filtered out if present.

```
class myservice (
  Enum['running', 'stopped'] $service_ensure,
  String                     $tempfile_contents,
  Optional[Array[String[1]]] $package_list = undef,
) {
```

- Rather than just saying that there was a type mismatch for `$package_list`, this example includes an additional assertion with an improved error message. The list can be "not given", or have an empty list of packages to install. An assertion is made that the list is an array of at least one String, and that the String is at least one character long.

```
  assert_type(Array[String[1], 1], $package_list) |$expected, $actual| {
    fail("Module ${module_name} does not support ${facts['os']['name']} as
 the list of packages is of type ${actual}")
  }

  package { $package_list:
    ensure => present,
  }

  file { "/tmp/${variable}":
    ensure   => present,
    contents => regsubst($tempfile_contents, '\d', '', 'G'),
    owner    => '0',
    group    => '0',
    mode     => '0644',
  }

  service { 'myservice':
    ensure    => $service_ensure,
    hasstatus => true,
  }

  Package[$package_list] -> Service['myservice']
```

```
}
```

In `hiera.yaml`: The default values can be merged if you want to extend with additional packages. If not, use `default_hierarchy` instead of `hierarchy`.

```
---
version: 5
defaults:
  data_hash: yaml_data

hierarchy:
- name: 'Per Operating System'
  path: "os/%{os.name}.yaml"
- name: 'Common'
  path: 'common.yaml'
```

In `data/common.yaml`:

```
myservice::service_ensure: running
```

In `data/os/centos.yaml`:

```
myservice::package_list:
- 'myservice-centos-package'
```

In `data/os/solaris.yaml`:

```
myservice::package_list:
- 'myservice-solaris-package1'
- 'myservice-solaris-package2'
```

### Public and private

Split your module into public and private classes and defined types where possible. Public classes or defined types should contain the parts of the module meant to be configured or customized by the user, while private classes should contain things you do not expect the user to change via parameters. Separating into public and private classes or defined types helps build reusable and readable code.

Help indicate to the user which classes are which by making sure all public classes have complete comments and denoting public and private classes in your documentation. Use the documentation tags “@api private” and “@api public” to make this clear. For complete documentation recommendations, see the Modules section.

### Chaining arrow syntax

Most of the time, use relationship metaparameters rather than chaining arrows. When you have many interdependent or order-specific items, chaining syntax may be used. A chain operator should appear on the same line as its right-hand operand. Chaining arrows must be used left to right.

Good: Points left to right:

```
Package['httpd'] -> Service['httpd']
```

Good: On the line of the right-hand operand:

```
Package['httpd']
-> Service['httpd']
```

Bad: Arrows are not all pointing to the right:

```
Service['httpd'] <- Package['httpd']
```

Bad: Must be on the right-hand operand's line:

```
Package['httpd'] ->
Service['httpd']
```

### Nested classes or defined types

Don't define classes and defined resource types within other classes or defined types. Declare them as close to node scope as possible. If you have a class or defined type which requires another class or defined type, put graceful failures in place if those required classes or defined types are not declared elsewhere.

Bad:

```
class apache {
  class ssl { ... }
}
```

Bad:

```
class apache {
  define config() { ... }
}
```

### Display order of parameters

In parameterized class and defined resource type definitions, you can list required parameters before optional parameters (that is, parameters with defaults). Required parameters are parameters that are not set to anything, including undef. For example, parameters such as passwords or IP addresses might not have reasonable default values.

You can also group related parameters, order them alphabetically, or in the order you encounter them in the code. How you order parameters is personal preference.

Note that treating a parameter like a namevar and defaulting it to `$title` or `$name` does not make it a required parameter. It should still be listed following the order recommended here.

Good:

```
class dhcp (
  $dnsdomain,
  $nameservers,
  $default_lease_time = 3600,
  $max_lease_time     = 86400,
) {}
```

Bad:

```
class ntp (
  $options   = "iburst",
  $servers,
  $multicast = false,
) {}
```

### Parameter defaults

Adding default values to the parameters in classes and defined types makes your module easier to use. As of Puppet 4.9.0, use Hiera data in the module and rely on automatic parameter lookup for class parameters. See the documentation about automatic parameter lookup for detailed information.

For versions earlier than Puppet 4.9.0, use the "params.pp" pattern. In simple cases, you can also specify the default values directly in the class or defined type.

Take care to declare the data type of parameters, as this provides automatic type assertions.

Good: Parameter defaults provided via APL > puppet 4.9.0:

```
class my_module (
  String $source,
  String $config,
) {
  # body of class
}
```

with a `hiera.yaml` in the root of the module:

```
---
version: 5
default_hierarchy:
- name: 'defaults'
  path: 'defaults.yaml'
  data_hash: yaml_data
```

and with the file `data/defaults.yaml`:

```
my_module::source: 'default source value'
my_module::config: 'default config value'
```

This places the values in the defaults hierarchy, which means that the defaults are not merged into overriding values. If you want to merge the defaults into those values, change the default_hierarchy to hierarchy.

Puppet 4.8 and earlier: Using `params.pp` pattern < Puppet 4.9.0:

```
class my_module (
  String $source = $my_module::params::source,
  String $config = $my_module::params::config,
) inherits my_module::params {
  # body of class
}
```

### Exported resources

Exported resources should be opt-in rather than opt-out. Your module should not be written to use exported resources to function by default unless it is expressly required.

When using exported resources, name the property `collect_exported`.

Exported resources should be exported and collected selectively using a search expression, ideally allowing user-defined tags as parameters so tags can be used to selectively collect by environment or custom fact.

Good:

```
define haproxy::frontend (
  $ports              = undef,
  $ipaddress          = [$::ipaddress],
  $bind               = undef,
  $mode               = undef,
```

```
  $collect_exported = false,
  $options          = {
    'option' => [
      'tcplog',
    ],
  },
) {
  # body of define
}
```

#### Parameter indentation and alignment

Parameters to classes or defined types must be uniformly indented in two spaces from the title. The equals sign should be aligned.

Good:

```
class profile::myclass (
  $var1    = 'default',
  $var2    = 'something else',
  $another = 'another default value',
) {

}
```

Good:

```
class ntp (
  Boolean                        $broadcastclient = false,
  Optional[Stdlib::Absolutepath] $config_dir      = undef,
  Enum['running', 'stopped']     $service_ensure  = 'running',
  String                         $package_ensure  = 'present',
  # ...
) {
# ...
}
```

Bad: Too many level of indentation:

```
class profile::myclass (
      $var1    = 'default',
      $var2    = 'something else',
      $another = 'another default value',
) {

}
```

Bad: No indentation:

```
class profile::myclass (
$var1    = 'default',
$var2    = 'something else',
$another = 'another default value',
) {

}
```

Bad: Misaligned equals sign:

```
class profile::myclass (
  $var1 = 'default',
```

```
  $var2   = 'something else',
  $another = 'another default value',
) {

}
```

When using if/else statements, align in the following way:

```
    if $something {
       $var = 'hour'
     } elsif $something_else {
       $var = 'minute'
     } else {
       $var = 'second'
     }
```

For more information on if/else statements, see Conditional statements and expressions.

### Class inheritance

In addition to scope and organization, there are some additional guidelines for handling classes in your module.

Don't use class inheritance; use data binding instead of `params.pp` pattern. Inheritance is used only for `params.pp`, which is not recommended in Puppet 4.

If you use inheritance for maintaining older modules, do not use it across module namespaces. To satisfy cross-module dependencies in a more portable way, include statements or relationship declarations. Only use class inheritance for `myclass::params` parameter defaults. Accomplish other use cases by adding parameters or conditional logic.

Good:

```
class ssh { ... }

class ssh::client inherits ssh { ... }

class ssh::server inherits ssh { ... }
```

Bad:

```
class ssh inherits server { ... }

class ssh::client inherits workstation { ... }

class wordpress inherits apache { ... }
```

### Public modules

When declaring classes in publicly available modules, use `include`, `contain`, or `require` rather than class resource declaration. This avoids duplicate class declarations and vendor lock-in.

### Type signatures

We recommend always using type signatures for class and defined type parameters. Keep the parameters and = signs aligned.

When dealing with very long type signatures, you can define type aliases and use short definitions. Good naming of aliases can also serve as documentation, making your code easier to read and understand. Or, if necessary, you can turn the 140 line character limit off. For more information on type signatures, see the `Type` data type.

#### Related information

Modules on page 200
Develop your module using consistent code and module structures to make it easier to update and maintain.

## Variables

Reference variables in a clear, unambiguous way that is consistent with the Puppet style.

### Referencing facts

When referencing facts, prefer the `$facts` hash to plain top-scope variables (such as `$::operatingsystem`).

Although plain top-scope variables are easier to write, the `$facts` hash is clearer, easier to read, and distinguishes facts from other top-scope variables.

### Namespacing variables

When referencing top-scope variables other than facts, explicitly specify absolute namespaces for clarity and improved readability. This includes top-scope variables set by the node classifier and in the main manifest.

This is not necessary for:

- the `$facts` hash.
- the `$trusted` hash.
- the `$server_facts` hash.

These special variable names are protected; because you cannot create local variables with these names, they always refer to top-scope variables.

Good:

```
$facts['operatingsystem']
```

Bad:

```
$::operatingsystem
```

Very bad:

```
$operatingsystem
```

### Variable format

When defining variables you must only use numbers, lowercase letters, and underscores. Do not use upper-case letters within a word, such as “CamelCase”, as it introduces inconsistency in style. You must not use dashes, as they are not syntactically valid.

Good:

```
$server_facts
$total_number_of_entries
$error_count123
```

Bad:

```
$serverFacts
$totalNumberOfEntries
$error-count123
```

## Conditionals

Conditional statements should follow Puppet code guidelines.

### Simple resource declarations

Avoid mixing conditionals with resource declarations. When you use conditionals for data assignment, separate conditional code from the resource declarations.

**Good:**

```
$file_mode = $facts['operatingsystem'] ? {
  'debian' => '0007',
  'redhat' => '0776',
   default => '0700',
}

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

**Bad:**

```
file { '/tmp/readme.txt':
  ensure  => file,
  content => "Hello World\n",
  mode    => $facts['operatingsystem'] ? {
    'debian' => '0777',
    'redhat' => '0776',
    default  => '0700',
  }
}
```

### Defaults for case statements and selectors

Case statements must have default cases. If you want the default case to be "do nothing," you must include it as an explicit `default: {}` for clarity's sake.

Case and selector values must be enclosed in quotation marks.

Selectors should omit default selections only if you explicitly want catalog compilation to fail when no value matches.

**Good:**

```
case $facts['operatingsystem'] {
  'centos': {
    $version = '1.2.3'
  }
  'solaris': {
    $version = '3.2.1'
  }
  default: {
    fail("Module ${module_name} is not supported on ${::operatingsystem}")
  }
}
```

When setting the default case, keep in mind that the default case should cause the catalog compilation to fail if the resulting behavior cannot be predicted on the platforms the module was built to be used on.

## Modules

Develop your module using consistent code and module structures to make it easier to update and maintain.

### Versioning

Your module must be versioned, and have metadata defined in the `metadata.json` file.

We recommend semantic versioning.

Semantic versioning, or SemVer, means that in a version number given as x.y.z:

- An increase in 'x' indicates major changes: backwards incompatible changes or a complete rewrite.
- An increase in 'y' indicates minor changes: the non-breaking addition of new features.
- An increase in 'z' indicates a patch: non-breaking bug fixes.

### Module metadata

Every module must have metadata defined in the `metadata.json` file.

Your metadata should follow the following format:

```
{
  "name": "examplecorp-mymodule",
  "version": "0.1.0",
  "author": "Pat",
  "license": "Apache-2.0",
  "summary": "A module for a thing",
  "source": "https://github.com/examplecorp/examplecorp-mymodule",
  "project_page": "https://github.com/examplecorp/examplecorp-mymodule",
  "issues_url": "https://github.com/examplecorp/examplecorp-mymodules/
issues",
  "tags": ["things", "stuff"],
  "operatingsystem_support": [
    {
      "operatingsystem":"RedHat",
      "operatingsystemrelease": [
        "5.0",
        "6.0"
      ]
    },
    {
      "operatingsystem": "Ubuntu",
      "operatingsystemrelease": [
        "12.04",
        "10.04"
     ]
    }
  ],
  "dependencies": [
    { "name": "puppetlabs/stdlib", "version_requirement": ">= 3.2.0
 <5.0.0" },
    { "name": "puppetlabs/firewall", "version_requirement": ">= 0.4.0
 <5.0.0" },
  ]
}
```

For additional information regarding the `metadata.json` format, see Adding module metadata in metadata.json.

### Dependencies

Hard dependencies must be declared explicitly in your module’s metadata.json file.

Soft dependencies should be called out in the README.md, and must not be enforced as a hard requirement in your metadata.json. A soft dependency is a dependency that is only required in a specific set of use cases. For an example, see the rabbitmq module.

Your hard dependency declarations should not be unbounded.

### README

Your module should have a README in `.md` (or `.markdown`) format. READMEs help users of your module get the full benefit of your work.

The Puppet README template offers a basic format you can use. If you create modules with Puppet Development Kit or the `puppet module generate` command, the generated README includes the template. Using the .md/.markdown format allows your README to be parsed and displayed by Puppet Strings, GitHub, and the Puppet Forge.

There's a thorough, detailed guide to writing a great README, but in general your README should:

- Summarize what your module does.
- Note any setup requirements or limitations, such as "This module requires the `puppetlabs-apache` module and only works on Ubuntu."
- Note any part of a user's system the module might impact (for example, "This module overwrites everything in `animportantfile.conf`.").
- Describe  how to customize and configure the module.
- Include usage examples and code samples for the common use cases for your module.

### Documenting Puppet code

Use Puppet Strings code comments to document your Puppet classes, defined types, functions, and resource types and providers. Strings processes the README and comments from your code into HTML or JSON format documentation. This allows you and your users to generate detailed documentation for your module.

Include comments for each element (classes, functions, defined types, parameters, and so on) in your module. If used, comments must precede the code for that element. Comments should contain the following information, arranged in this order:

- A description giving an overview of what the element does.
- Any additional information about valid values that is not clear from the data type. For example, if the data type is `[String]`, but the value must specifically be a path.
- The default value, if any, for that element,

Multiline descriptions must be uniformly indented by at least one space:

```
# @param config_epp Specifies a file to act as a EPP template for the config
 file.
#  Valid options: a path (absolute, or relative to the module path). Example
 value:
#  'ntp/ntp.conf.epp'. A validation error is thrown if you supply both this
 param **and**
#  the `config_template` param.
```

If you use Strings to document your module, include information about Strings in the Reference section of your README so that your users know how to generate the documentation. See Puppet Strings documentation for details on usage, installation, and correctly writing documentation comments.

If you do not include Strings code comments, you should include a Reference section in your README with a complete list of all classes, types, providers, defined types, and parameters that the user can configure. Include a brief description, the valid options, and the default values (if any).

For example, this is a parameter for the `ntp` module's `ntp` class: `package_ensure`:

```
Data type: String.

Whether to install the NTP package, and what version to install. Values:
 'present', 'latest', or a specific version number.

Default value: 'present'.
```

For more details and examples, see the module documentation guide.

### CHANGELOG

Your module should include a change log file called `CHANGELOG.md` or `.markdown`. Your change log should:

- Have entries for each release.
- List bugfixes and features included in the release.
- Specifically call out backwards-incompatible changes.

### Examples

In the `/examples` directory, include example manifests that demonstrate major use cases for your module.

```
modulepath/apache/examples/{usecase}.pp
```

The example manifest should provide a clear example of how to declare the class or defined resource type. It should also declare any classes required by the corresponding class to ensure `puppet apply` works in a limited, standalone manner.

### Testing

Use one or more of the following community tools for testing your code and style:

- puppet-lint tests your code for adherence to the style guidelines.
- metadata-json-lint tests your `metadata.json` for adherence to the style guidelines.
- For testing your module, we recommend rspec. rspec-puppet can help you write rspec tests for Puppet.

## Files and paths on Windows

Puppet and Windows handle directory separators and line endings in files somewhat differently, so you must be aware of the differences when you are writing manifests to manage Windows systems.

### Directory separators in file paths

Several resource types (including `file`, `exec`, and `package`) take file paths as values for various attributes. The Puppet language uses the backslash (\) as an escape character in quoted strings. However, Windows also uses the backslash to separate directories in file paths, such as `C:\Program Files\PuppetLabs`. Additionally, Windows file system APIs accept both backslashes and forward slashes in file paths, but some Windows programs accept only backslashes.

Generally, if Puppet itself is interpreting the file path, or if the file path is meant for the master, use forward slashes. If the file path is being passed directly to a Windows program, use backslashes. The following table lists common directory path uses and what kind of slashes are required by each.

| File path usage | Slash type |
|---|---|
| Template paths, such as `template('my_module/content.erb')`. | Forward slash (/) |

| File path usage | Slash type |
|---|---|
| `puppet:///` URLs. | Forward slash (/) |
| The `path` attribute or title of a `file` resource. | Forward slash (/) or backslash (\) |
| The `source` attribute of a `package` resource. | Forward slash (/) or backslash (\) |
| Local paths in a `file` resource's `source` attribute. | Forward slash (/) or backslash (\) |
| The `command` of an `exec` resource. However, some executables, such as `cmd.exe`, require backslashes. | Forward slash (/) or backslash (\) |
| Any file paths included in the `command` of a `scheduled_task` resource. | Backslash (\) |
| Any file paths included in the `install_options` of a `package` resource. | Backslash (\) |
| Any file paths used for Windows PowerShell DSC resources. For these resources, single quote strings whenever possible. | Backslash (\) |

### Line endings in files

Windows uses CRLF line endings instead of *nix's LF line endings. You should be aware of a few issues:

- If you specify the contents of a file with the `content` attribute, Puppet writes the content in binary mode. To create files with CRLF line endings, specify the `\r\n` escape sequence as part of the content.
- When downloading a file to a Windows node with the `source` attribute, Puppet transfers the file in binary mode, leaving the original newlines untouched.
- If you are using version control, such as Git, ensure that it is configured to use CRLF line endings.
- Non-`file` resource types that make partial edits to a system file, such as the `host` resource type, which manages the `%windir%\system32\drivers\etc\hosts` file, manage their files in text mode and automatically translate between Windows and *nix line endings.

  **Note:** When writing your own resource types, you can get this behavior by using the `flat` file type.

### Related information

Templates on page 336
Templates are documents that combine code, data, and literal text to produce a final rendered output. The goal of a template is to manage a complicated piece of text with simple inputs.

Strings on page 295
Strings are unstructured text fragments of any length. They’re a common and useful data type.

## Code comments

To add comments to your Puppet code, use shell-style or hash comments.

Hash comments begin with a hash symbol (#) and continue to the end of a line. You can start comments either 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,
}
```

## Variables

Variables store values so that those values can be accessed in code later.

After you've assigned a variable a value, you cannot reassign it. Variables depend on order of evaluation: you must assign a variable a value before it can be resolved.

**Note:** Puppet contains built-in variables that you can use in your manifests. For a list of these, see the page on facts and built-in variables.

### Assigning variables

Prefix variable names with a dollar sign (`$`). Assign values to variables with the equal sign (`=`) assignment operator.

Variables accept values of any data type. You can assign literal values, or you can assign any statement that resolves to a normal value, including expressions, functions, and other variables. The variable then contains the value that the statement resolves to, rather than a reference to the statement.

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

Assign variables using their short name within their own scope. You cannot assign values in one scope from another scope. For example, you can assign a value to the `apache::params` class's `$vhostdir` variable only from within the `apache::params` class.

You can assign multiple variables at the same time from an array or hash.

To assign multiple variables from an array, you must specify an equal number of variables and values. If the number of variables and values do not match, the operation fails. You can also use nested arrays. For example, all of the variable assignments shown below are valid.

```
[$a, $b, $c] = [1,2,3]      # $a = 1, $b = 2, $c = 3
[$a, [$b, $c]] = [1,[2,3]]  # $a = 1, $b = 2, $c = 3
[$a, $b] = [1, [2]]            # $a = 1, $b = [2]
[$a, [$b]] = [1, [2]]          # $a = 1, $b = 2
```

When you assign multiple variables with a hash, list the variables in an array on the left side of the assignment operator, and list the hash on the right. Hash keys must match their corresponding variable name. This example assigns a value of 10 to the `$a` variable and a value of 20 to the `$b` variable.

```
[$a, $b] = {a => 10, b => 20}  # $a = 10, $b = 20
```

You can include extra key-value pairs in the hash, but all variables to the left of the operator must have a corresponding key in the hash:

```
[$a, $c] = {a => 5, b => 10, c => 15, d => 22}  # $a = 5, $c = 15
```

Puppet allows a given variable to be assigned a value only one time within a given scope. This is a little different from most programming languages. You cannot change the value of a variable, but you can 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" )
}
```

#### Resolution

You can use the name of a variable in any place where a value of the variable's data type would be accepted, including expressions, functions, and resource attributes. Puppet replaces the name of the variable with its value. By default, unassigned variables have a value of `undef`. See the section about unassigned variables and strict mode for more details.

In these examples, the `content` parameter value resolves to whatever value has been assigned to the `$content` variable. The `$address_array` variable resolves to an array of the values assigned to the `$address1`, `$address2`, and `$address3` variables:

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

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

#### Interpolation

Puppet can resolve variables that are included in double-quoted strings; this is called *interpolation*. Inside a double-quoted string, surround the name of the variable (the portion after the `$`) with curly braces, such as `${var_name}`. This syntax is optional, but it helps to avoid ambiguity and allows variables to be placed directly next to non-whitespace characters. These optional curly braces are permitted only inside strings.

For example, the curly braces make it easier to quickly identify the variable `$homedir`.

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

#### Scope

The area of code where a given variable is visible is dictated by its *scope*. Variables in a given scope are available only 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.

You can access out-of-scope variables from named scopes by using their qualified names, which include namespaces representing the variable's scope. The scope is where the variable is defined and assigned a value. For example, the qualified name of this `$vhost` variable shows that the variable is found and assigned a value in the `apache::params` class:

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

Variables can be assigned outside of any class, type, or node definition. These *top scope* variables have an empty string as their first namespace segment, so that the qualified name of a top scope variable begins with a double colon, such as `$::osfamily`.

### Unassigned variables and strict mode

By default, you can access variables that have never had values assigned to them. If you do, their value is `undef`. This can be a problem, because an unassigned variable is often an accident or a typo. To make unassigned variable usage return an error, so that you can find and fix the problem, enable strict mode by setting `strict_variables = true` strict_variables in the `puppet.conf` file on your master and on any nodes that run `puppet apply`. For details about this setting, see the configuration page.

### Related information

Expressions and operators on page 232
*Expressions* are statements that resolve to values. You can use expressions almost anywhere a value is required. Expressions can be compounded with other expressions, and the entire combined expression resolves to a single value.

Function calls on page 249
Functions are plug-ins, written in Ruby, that you can call during catalog compilation. A call to any function is an expression that resolves to a value. Most functions accept one or more values as arguments, and return a resulting value.

## Naming variables

Some variable names are reserved; for detailed information, see the reserved name page.

### Variable names

Variable names are case-sensitive and must begin with a dollar sign (`$`). Most variable names must start with a lowercase letter or an underscore. The exception is regex capture variables, which are named with only numbers.

Variable names can include:

- Uppercase and lowercase letters
- Numbers
- Underscores ( _ ). If the first character is an underscore, access that variable only from its own local scope.

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`.

Optionally, the name of the very first namespace can be empty, representing the top namespace. The main reason to namespace this way is to indicate to anyone reading your code that you're accessing a top-scope variable, such as `$::is_virtual`.

You can also use a regular expression for variable names. Short variable names match the following regular expression:

```
\A\$[a-z0-9_][a-zA-Z0-9_]*\Z
```

Qualified variable names match the following regular expression:

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

## Resources

Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.

Resources contained in classes and defined types share the relationships of those classes and defined types. Resources are not subject to scope: a resource in any area of code can be referenced from any other area of code.

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:

1. Reads the actual state of the resource on the target system.
2. Compares the actual state to the desired state.
3. If necessary, changes the system to enforce the desired state.
4. Logs any changes made to the resource. These changes appear in Puppet agent's log and in the run report, which is sent to the master and forwarded to any specified report processors.

If the catalog doesn't contain a particular resource, Puppet does nothing with whatever that resource described. If you remove a package resource from your manifests, Puppet doesn't uninstall the package; instead, it just ignores it. To remove a package, manage it as a resource and set `ensure => absent`.

You can delay adding resources to the catalog. For example, classes and defined types can contain groups of resources. These resources are managed only if you add that class or defined resource to the catalog. Virtual resources are added to the catalog only after they are realized.

### Resource declarations

At minimum, every resource declaration has a *resource type*, a *title*, and a set of *attributes*:

```
<TYPE> { '<TITLE>':
  <ATTRIBUTE> => <VALUE>,
}
```

The resource title and attributes are called the resource body. A resource declaration can have one resource body or multiple resource bodies of the same resource type.

Resource declarations are expressions in the Puppet language — they always have a side effect of adding a resource to the catalog, but they also resolve to a value.The value of a resource declaration is an array of *resource references*, with one reference for each resource the expression describes.

A resource declaration has extremely low precedence; in fact, it's even lower than the variable assignment operator (`=`). This means that if you use a resource declaration for its value, you must surround it with parentheses to associate it with the expression that uses the value.

If a resource declaration includes more than one resource body, it declares multiple resources of that resource type. The resource body is a title and a set of attributes; each body must be separated from the next one with a semicolon. Each resource in a declaration is almost completely independent of the others, and they can have completely different values for their attributes. The only connections between resources that share an expression are:

- They all have the same resource type.
- They can all draw from the same pool of default values, if a resource body with the title `default` is present.

### Resource uniqueness

Each resource must be unique; 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 resource `title` and the `name` attribute or *namevar* to identify duplicate resources — if either the title or the name is duplicated within a given resource type, catalog compilation fails. See the page about resource syntax for details about resource titles and namevars. To provide the same resource for multiple classes, use a class or a virtual resource to add it to the catalog in multiple places without duplicating it. See classes and virtual resources for more information.

### Relationships and ordering

By default, Puppet applies unrelated resources in the order in which they're written in the manifest. If a resource must be applied before or after some other resource, you should declare a relationship between them, to show that their order isn't coincidental. You can also make changes in one resource cause a refresh of some other resource. See the Relationships and ordering page for more information.

Otherwise, you can customize the default order in which Puppet applies resources with the ordering setting. See the configuration page for details about this setting.

### Resource types

Every resource is associated with a *resource type*, which determines the kind of configuration it manages. Puppet has built-in resource types such as `file`, `service`, and `package`. See the resource type reference for a complete list and information about the built-in resource types.

You can also add new resource types to Puppet:

- Defined types are lightweight resource types written in the Puppet language.
- Custom resource types are written in Ruby and have the same capabilities as Puppet's built-in types.

### Title

A resource's title is a string that uniquely identifies the resource to Puppet. In a resource declaration, the title is the identifier after the first curly brace and before the colon. For example, in this file resource declaration, the title is `/etc/passwd`:

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

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

The title of a resource differs from the *namevar* of the resource. Whereas the title identifies the resource to Puppet itself, the namevar identifies the resource to the target system and is usually specified by the resource's `name` attribute. The resource title doesn't have to match the namevar, but you'll often want it to: the value of the namevar attribute defaults to the title, so using the name in the title can save you some typing.

If a resource type has multiple namevars, the type specifies whether and how the title maps to those namevars. For example, the `package` type uses the `provider` attribute to help determine uniqueness, but that attribute has no special relationship with the title. See each type's documentation for details about how it maps title to namevars.

### Attributes

Attributes describe the desired state of the resource; each attribute handles some aspect of the resource.For example, the `file` type has a `mode` attribute that specifies the permissions for the file.

Each resource type has its own set of available attributes; see the resource type reference for a complete list. Most resource types have a handful of crucial attributes and a larger number of optional ones.Attributes accept certain data types, such as strings, numbers, hashes, or arrays. Each attribute that you declare must have a value. Most attributes are optional, which means they have a default value, so you do not have to assign a value. If an attribute has no default, it is considered required, and you must assign it a value.

Most resource types contain an `ensure` attribute. This attribute generally manages the most basic state of the resource on the target system, such as whether a file exists, whether a service is running or stopped, or whether a package is installed or uninstalled.The values accepted for the `ensure` attribute vary by resource type. Most accept `present` and `absent`, but there are variations. Check the reference for each resource type you are working with.

**Tip:** Resource and type attributes are sometimes referred to as parameters. Puppet also has properties, which are slightly different from parameters: properties correspond to something measurable on the target system, whereas parameters change how Puppet manages a resource. A property always represents a concrete state on the target system. When talking about resource declarations in Puppet, parameter is a synonym for attribute.

### Namevars and name

Every resource on a target system must have a unique identity; you cannot have two services, for example, with the same name. This identifying attribute in Puppet is known as the *namevar*.

Each resource type has an attribute that is designated to serve as the namevar. For most resource types, this is the `name` attribute, but some types use other attributes, such as the `file` type, which uses `path`, the file's location on disk, for its namevar. If a type's namevar is an attribute other than `name`, this is listed in the type reference documentation.

Most types have only one namevar. With a single namevar, the value must be unique per resource type. There are a few rare exceptions to this rule, such as the `exec` type, where the namevar is a command. However, some resource types, such as `package`, have multiple namevar attributes that create a composite namevar. For example, both the `yum` provider and the `gem` provider have `mysql` packages, so both the `name` and the `provider` attributes are namevars, and Puppet uses both to identify the resource.

The namevar differs from the resource's *title*, which identifies a resource to Puppet's compiler rather than to the target system. In practice, however, a resource's namevar and the title are often the same, because the namevar usually defaults to the title. If you don't specify a value for a resource's namevar when you declare the resource, Puppet uses the resource's title.

You might want to specify different a namevar that is different from the title when you want a consistently titled resource to manage something that has different names on different platforms. For example, the NTP service might be `ntpd` on Red Hat systems, but `ntp` on Debian and Ubuntu. You might title the service "ntp," but set its namevar --- the `name` attribute --- according to the operating system. Other resources can then form relationships to the resource without the title changing.

### Metaparameters

Some attributes in Puppet can be used with every resource type. These are called *metaparameters*. These don't map directly to system state. Instead, metaparameters affect Puppet's behavior, usually specifying the way in which resources relate to each other.

The most commonly used metaparameters are for specifying order relationships between resources. See the documentation on relationships and ordering for details about those metaparameters. See the full list of available metaparameters in the metaparameter reference.

## Resource syntax

You can accomplish a lot with just a few resource declaration features, or you can create more complex declarations that do more.

### Basic syntax

The simplified form of a resource declaration includes:

- The resource type, which is a word with no quotes.
- 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 lowercase word with no quotes.
  - A `=>` (called an arrow, "fat comma," or "hash rocket").
  - A value, which can have any [data type][datatype].
  - A trailing comma.
- A closing curly brace (`}`).

You can use any amount of whitespace in the Puppet language.

This example declares a file resource with the title `/etc/passwd`. This declaration's `ensure` attribute ensures that the specified file is created, if it does not already exist on the node. The rest of the declaration sets values for the file's `owner`, `group`, and `mode` attributes.

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

### Complete syntax

By creating more complex resource declarations, you can:

- Describe many resources at once.
- Set a group of attributes from a hash with the `*` attribute.
- Set default attributes.
- Specify an abstract resource type.
- Amend or override attributes after a resource is already declared.

The complete generalized form of a resource declaration expression is:

- The resource type, which can be one of:
  - A lowercase word with no quotes, such as `file`.
  - A resource type data type, such as `File`, `Resource[File]`, or `Resource['file']`. It must have a type but not a title.
- An opening curly brace (`{`).
- One or more resource bodies, separated with semicolons (`;`). Each resource body consists of:
  - A title, which can be one of:
    - A string.
    - An array of strings, which declares multiple resources.
    - The special value `default`, which sets default attribute values for other resource bodies in the same expression.
  - A colon (`:`).
  - Optionally, any number of attribute and value pairs, separated with commas (`,`). Each attribute/value pair consists of:
    - An attribute name, which can be one of:
      - A lowercase word with no quotes.
      - The special attribute `*`, called a "splat," which takes a hash and sets other attributes.
      - A `=>`, called an arrow, a "fat comma," or a "hash rocket".
      - A value, which can have any data type.
    - Optionally, a trailing comma after the last attribute/value pair.
- Optionally, a trailing semicolon after the last resource body.

- A closing curly brace (})

```
<TYPE> {
  default:
    *                         => <HASH OF ATTRIBUTE/VALUE PAIRS>,
    <ATTRIBUTE> => <VALUE>,
  ;
  '<TITLE>':
    *                         => <HASH OF ATTRIBUTE/VALUE PAIRS>,
    <ATTRIBUTE> => <VALUE>,
  ;
  '<NEXT TITLE>':
    ...
  ;
  ['<TITLE'>, '<TITLE>', '<TITLE>']:
    ...
  ;
```

### Resource declaration default attributes

If a resource declaration includes a resource body with a title of `default`, Puppet doesn't create a new resource named "default." Instead, every other resource in that declaration uses attribute values from the `default` body if it doesn't have an explicit value for one of those attributes. This is also known as "per-expression defaults."

Resource declaration defaults are useful because it lets you set many attributes at once, but you can still override some of them.

This example declares several different files, all using the default values set in the `default` resource body. However, the `mode` value for the the files in the last array (`['ssh_config', 'ssh_host_dsa_key.pub'....`) is set explicitly instead of using the default.

```
file {
  default:
    ensure => file,
    owner  => "root",
    group  => "wheel",
    mode   => "0600",
  ;
  ['ssh_host_dsa_key', 'ssh_host_key', 'ssh_host_rsa_key']:
    # use all defaults
  ;
  ['ssh_config', 'ssh_host_dsa_key.pub', 'ssh_host_key.pub',
 'ssh_host_rsa_key.pub', 'sshd_config']:
    # override mode
    mode => "0644",
  ;
}
```

The position of the `default` body in a resource declaration doesn't matter; resources above and below it all use the default attributes if applicable.You can only have one `default` resource body per resource declaration.

### Setting attributes from a hash

You can set attributes for a resource by using the splat attribute, which uses the splat or asterisk character `*`, in the resource body.

The value of the splat (`*`) attribute must be a hash where:

- Each key is the name of a valid attribute for that resource type, as a string.
- Each value is a valid value for the attribute it's assigned to.

This sets values for that resource's attributes, using every attribute and value listed in the hash.

For example, the splat attribute in this declaration sets the `owner`, `group`, and `mode` settings for the file resource.

```
$file_ownership = {
  "owner" => "root",
  "group" => "wheel",
  "mode"  => "0644",
}

file { "/etc/passwd":
  ensure => file,
  *      => $file_ownership,
}
```

You cannot set any attribute more than once for a given resource; if you try, Puppet raises a compilation error. This means that:

- If you use a hash to set attributes for a resource, you cannot set a different, explicit value for any of those attributes. For example, if `mode` is present in the hash, you can't also set `mode => "0644"` in that resource body.
- You can't use the `*` attribute multiple times in one resource body, since the splat itself is an attribute.

To use some attributes from a hash and override others, either use a hash to set per-expression defaults, as described in the section on resource declaration defaults, or use the merging operator, `+` to combine attributes from two hashes, with the right-hand hash overriding the left-hand one.

### Abstract resource types

Because a resource declaration can accept a resource type data type as its resource type , you can use a `Resource[<TYPE>]` value to specify a non-literal resource type, where the `<TYPE>` portion can be read from a variable.That is, the following three examples are equivalent to each other:

```
file { "/tmp/foo": ensure => file, }
File { "/tmp/foo": ensure => file, }
Resource[File] { "/tmp/foo": ensure => file, }
```

```
$mytype = File
Resource[$mytype] { "/tmp/foo": ensure => file, }
```

```
$mytypename = "file"
Resource[$mytypename] { "/tmp/foo": ensure => file, }
```

This lets you declare resources without knowing in advance what type of resources they'll be, which can enable transformations of data into resources. For a demonstration, see the `create_resources` example below.

### Arrays of titles

If you specify an array of strings as the title of a resource body, Puppet creates multiple resources with the same set of attributes. This is useful when you have many resources that are nearly identical.

For example:

```
$rc_dirs = [
  '/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 { $rc_dirs:
  ensure => directory,
```

```
  owner  => 'root',
  group  => 'root',
  mode   => '0755',
}
```

If you do this, you must let the namevar attributes of these resources default to their titles. You can't specify an explicit value for the namevar, because it applies to all of the resources.

### Adding or modifying attributes

Although you cannot declare the same resource twice, you can add attributes to an resource that has already been declared. In certain circumstances, you can also override attributes. You can amend attributes with either a resource reference, a collector, or from a hash using the splat (`*`) attribute.

To amend attributes with the splat attribute, see the section about setting attributes from a hash.

To amend attributes with a resource reference, add a resource reference attribute block to the resource that's already declared. Normally, you can only use resource reference blocks to add previously unmanaged attributes to a resource; it cannot override already-specified attributes. The general form of a resource reference attribute block is:

- A resource reference to the resource in question
- An opening curly brace
- Any number of attribute `=>` value pairs
- A closing curly brace

For example, this resource reference attribute block amends values for the `owner`, `group`, and `mode` attributes:

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

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

You can also amend attributes with a collector.

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

For resource attributes that accept multiple values in an array, such as the relationship metaparameters, you can add to the existing values instead of replacing them by using the "plusignment" (`+>`) keyword instead of the usual hash rocket (`=>`). For details, see appending to attributes in the classes documentation.

This example amends the `owner`, `group`, and `mode` attributes of any resources that match the collector:

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

include base::linux

File <| tag == 'base::linux' |> {
 owner => 'root',
```

```
  group => 'root',
  mode => '0640',
}
```

**CAUTION:** Be very careful when amending attributes with a collector. Test with `--noop` to see what changes your code would make.

- It can override other attributes you've already specified, regardless of class inheritance.
- It can affect large numbers of resources at one time.
- It implicitly realizes any virtual resources the collector matches.
- Because it ignores class inheritance, it can override the same attribute more than one time, which results in an evaluation order race where the last override wins.

### Local resource defaults

Because resource default statements are subject to dynamic scope, you can't always tell what areas of code will be affected. Generally, you should not include classic resource default statements anywhere other than in your site manifest (`site.pp`). See the resource defaults documentation for details. Whenever possible, use resource declaration defaults, also known as per-expression defaults.

However, resource default statements can be powerful, allowing you to set important defaults, such as file permissions, across resources. Setting local resource defaults is a way to protect your classes and defined types from accidentally inheriting defaults from classic resource default statements.

To set local resource defaults, define your defaults in a variable and re-use them in multiple places, by combining resource declaration defaults and setting attributes from a hash.

This example defines defaults in a `$file_defaults` variable, and then includes the variable in a resource declaration default with a hash.

```
class mymodule::params {
  $file_defaults = {
    mode  => "0644",
    owner => "root",
    group => "root",
  }
  # ...
}

class mymodule inherits mymodule::params {
  file { default: *=> $mymodule::params::file_defaults;
    "/etc/myconfig":
      ensure => file,
    ;
  }
}
```

## Relationships and ordering

Puppet normally applies resources in the order they are declared in their manifest. To manage a group of resources in a specific order, explicitly declare such relationships with relationship metaparameters, chaining arrows, and the `require` function.

To override Puppet's default manifest ordering, declare an explicit relationship between resources. All relationships cause Puppet to manage specific resources before other resources. Relationships are not limited by evaluation-order; you can declare a relationship with a resource before that resource has been declared.

### Refreshing and notification

Some resource types can refresh when one of their dependencies changes. For example, some services must restart when their configuration files change, so `service` resources can refresh by restarting the service.

The built-in resource types that can refresh are `service`, `exec`, and `package`. For specific details about these types, see the resource reference.

To specify that a resource should refresh when a related resource changes, create a notifying relationship with the `subscribe` or `notify` metaparameters or the notification chaining arrow (`~>`). When a resources changes, it sends a refresh event to any resources that subscribe to it. Those resources that are subscribed receive the refresh event.

When receiving refresh events:

- If a resource gets a refresh event during a run, and its resource type has a refresh action, it performs that action.
- If a resource gets a refresh event, but its resource type cannot refresh, nothing happens.
- If a class or defined resource gets a refresh event, every resource it contains also gets a refresh event.
- A resource can perform its refresh action up to once per run. If it receives multiple refresh events, they're combined, and the resource refreshes only once.

When sending refresh events:

- If a resource is not in its desired state, and Puppet makes changes to it during a run, it sends a refresh event to any subscribed resources.
- If a resource performs its refresh action during a run, it sends a refresh event to any subscribed resources.
- If Puppet changes or refreshes any resource in a class or defined resource, that class or defined resource sends a refresh event to any subscribed resources.

If non-operational (no-op) mode is enabled:

- The resource does not refresh when it receives a refresh event. Instead, Puppet logs a message stating what would have happened.
- The resource does not send refresh events to subscribed resources. Instead, Puppet logs messages stating what would have happened to any resources further down the subscription chain.

For more information about refresh behavior, see the types documentation.

### Automatic relationships

Certain resource types can have automatic relationships with other resources, using autorequire, autonotify, autobefore, or autosubscribe. This creates an ordering relationship without you explicitly stating one.

Puppet establishes automatic relationships between types and resources when it applies a catalog. It searches the catalog for any resources that match certain rules and processes them in the correct order, sending refresh events if necessary. If any explicit relationship, such as those created by chaining arrows, conflicts with an automatic relationship, the explicit relationship take precedence. Puppet

### Missing dependencies

If one of the resources in a relationship is never declared, compilation fails 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 skips the subsequent resource and log the following messages:

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

It then continues to apply any unrelated resources. Any resources that depend on the skipped resource are also skipped. This helps prevent an inconsistent system state, rather than attempting to apply a resource that might have broken prerequisites.

### Dependency cycles

If two or more resources require each other in a loop, Puppet compiles the catalog but won't be able to apply it. Puppet logs an error like the following, and attempts to help 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`.

### Related information

Containment of resources on page 367
Containment enables you to control where and when specific parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types.

## Relationship metaparameters

You can use certain metaparameters to establish relationships by setting any of them as an attribute in any resource.

Set the value of any relationship metaparameter to either a resource reference or an array of references that point to one or more target resources:

- `before`: Applies a resource before the target resource.
- `require`: Applies a resource after the target resource.
- `notify`: Applies a resource before the target resource. The target resource refreshes if the notifying resource changes.
- `subscribe`: Applies a resource after the target resource. The subscribing resource refreshes 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 creates the same relationship. The same is true of notify and subscribe.

The two examples below create the same ordering relationship, ensuring that the `openssh-server` package is managed before the `sshd_config` file:

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

```
file { '/etc/ssh/sshd_config':
  ensure  => file,
  mode    => '0600',
  source  => 'puppet:///modules/sshd/sshd_config',
  require => Package['openssh-server'],
}
```

The two examples below create the same notifying relationship, so that if Puppet changes the `sshd_config` file, it sends a notification to the `sshd` service:

```
file { '/etc/ssh/sshd_config':
  ensure => file,
```

```
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
  notify => Service['sshd'],
}
```

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

Because an array of resource references can contain resources of differing types, these two examples also create the same ordering relationship. In both examples, Puppet manages the `openssh-server` package and the `sshd_config` file before it manages the `sshd` service.

```
service { 'sshd':
  ensure  => running,
  require => [
    Package['openssh-server'],
    File['/etc/ssh/sshd_config'],
  ],
}
```

```
package { 'openssh-server':
  ensure => present,
  before => Service['sshd'],
}

file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
  before => Service['sshd'],
}
```

**Related information**

Resources on page 207
Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.

Resource and class references on page 320
Resource references identify a specific Puppet resource by its type and title. Several attributes, such as the relationship metaparameters, require resource references.

The Array data type on page 310
The data type of arrays is `Array`. By default, `Array` matches arrays of any length, provided all values in the array match the abstract data type `Data`. You can use parameters to restrict which values `Array` matches.

## Chaining arrows

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

The ordering arrow is a hyphen and a greater-than sign (`->`). It applies the resource on the left before the resource on the right.

The notifying arrow is a tilde and a greater-than sign (`~>`). It applies the resource on the left first. If the left-hand resource changes, the right-hand resource refreshes.

In this example, Puppet applies configuration to the `ntp.conf` file resource and notifies the `ntpd` service resource if there are any changes.

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

**Note:** Most of the time, you should use relationship metaparameters, not chaining arrows. Metaparameters are more explicit and easier to maintain. See the Puppet language style guide for information on when and how to use chaining arrows.

### Operands

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

- Resource references, including multi-resource references.
- Arrays of resource references.
- Resource declarations.
- Resource collectors.

You can link operands to apply a series of relationships and notifications. In this example, Puppet applies configuration to the package, notifies the file resource if there are changes, and then, if there are resulting changes to the file resouce, Puppet notifies the service resource:

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

Resource declarations can be chained. That means you can use chaining arrows to make Puppet apply a section of code in the order that it is written. This example applies configuration to the package, the file, and the service, in that order, with each related resource notifying the next of any changes:

```
# first:
package { 'openssh-server':
  ensure => present,
} # and then:
-> file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
} # and then:
~> service { 'sshd':
  ensure => running,
  enable => true,
}
```

Collectors can also be chained, so you can create relationships between many resources at one time. This example applies all Yum repository resources before applying any package resources, which protects any packages that rely on custom repositorie :

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

### Capturing resource references for generated resources

In Puppet, the value of a resource declaration is a reference to the resource it creates.

This is useful if you're automatically creating resources whose titles you can't predict: use the iteration functions to declare several resources at once or use an array of strings as a resource title. If you assign the resulting resource references to a variable, you can then use them in chaining statements without ever knowing the final title of the affected resources.

For example:

- The `map` function iterates over its arguments and returns an array of values, with each value produced by the last expression in the block. If that last expression is a resource declaration, `map` produces an array of resource references, which you could then use as an operand for a chaining arrow.
- For a resource declaration whose title is an array, the value is itself an array of resource references that you can assign to a variable and use in a chaining statement.

### Cautions when chaining resource collectors

<table>
<tr><td>Chains can create dependency cycles.</td><td>Chained collectors can cause huge dependency cycles; be careful when using them. They can also be dangerous when used with virtual resources, which are implicitly realized by collectors.</td></tr>
<tr><td>Chains can break.</td><td>Although you can usually chain many resources or collectors together (<code>File['one'] -> File['two'] -> File['three']</code>), the chain can break if it includes a collector whose search expression doesn't match any resources.</td></tr>
<tr><td>Implicit properties aren't searchable.</td><td>Collectors can search only on attributes present in the manifests; they cannot see properties that are automatically set or are read from the target system. For example, the chain <code>Yumrepo <| |> -> Package <| provider == yum |></code>, creates only relationships with packages whose <code>provider</code> attribute is explicitly set to <code>yum</code> in the manifests. It would not affect packages that didn't specify a provider but use Yum because it's the operating system's default provider.</td></tr>
</table>

### 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. Avoid these reversed forms, as they are confusing and difficult to notice.

### Related information

Resource and class references on page 320
Resource references identify a specific Puppet resource by its type and title. Several attributes, such as the relationship metaparameters, require resource references.

Resources on page 207
Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.

Resource collectors on page 358
Resource collectors select a group of resources by searching the attributes of each resource in the catalog, even resources which haven’t yet been declared at the time the collector is written. Collectors realize virtual resources, are used in chaining statements, and override resource attributes. Collectors have an irregular syntax that enables them to function as a statement and a value.

Lambdas on page 355
Lambdas are blocks of Puppet code passed to functions. When a function receives a lambda, it provides values for the lambda’s parameters and evaluates its code. If you use other programming languages, think of lambdas as anonymous functions that are passed to other functions.

Virtual resources on page 360
A virtual resource declaration specifies a desired state for a resource without enforcing that state. Puppet manages the resource by realizing it elsewhere in your manifests. This divides the work done by a normal resource declaration into

two steps. Although virtual resources are declared one time, they can be realized any number of times, similar to a class.

### The `require` function

Use the `require` function to declare a class and make it a dependency of the surrounding container.

For example:

```
class wordpress {
  require apache
  require mysql
  ...
}
```

The above example causes 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, you can create more complex behavior by combining `include` and chaining arrows inside a class definition. This example notifies and restarts every service in the `apache::ssl` class if any of the SSL certificates on the node change:

```
class apache::ssl {
  include site::certificates
  Class['site::certificates'] ~> Class['apache::ssl']
}
```

## Classes

*Classes* are named blocks of Puppet code that are stored in modules and applied later when they are invoked by name. You can add classes to a node's catalog by either declaring them in your manifests or assigning them from an external node classifier (ENC). Classes generally configure large or medium-sized chunks of functionality, such as all of the packages, configuration files, and services needed to run an application.

### Defining classes

Defining a class makes it available for later use. It doesn't add any resources to the catalog — to do that, you must declare the class or assign it from an external node classifier (ENC).

Create a class by writing a class definition in a manifest (`.pp`) file. Store class manifests in the `manifests/` directory of a module. Define only one class in a manifest, and give the manifest file the same name as the class. Puppet automatically loads any classes that are present in a valid module. See module fundamentals to learn more about module structure and usage.

A class contains all of its resources. This means any relationships formed with the class as a whole is extended to every resource in the class. Every resource in a class gets automatically tagged with the class's name and each of its namespace segments.

Classes can contain other classes, but you must explicitly specify that a class should be contained with the `contain` function. For more information, see the documentation about containing classes. A contained class is automatically tagged with the name of its container.

**Tip:** Unlike many parts of Puppet code, class definitions aren't expressions, so you can't use them where a value is expected.

The general form of a class definition is:

- The `class` keyword.
- The name of the class.

- An optional parameter list, which consists of:
  - An opening parenthesis.
  - A comma-separated list of parameters, such as`String $myparam = "value"`. Each parameter consists of:
    - An optional data type, which restricts the allowed values for the parameter. If not specified, the data type defaults to `Any`.
    - A variable name to represent the parameter, including the dollar sign (`$`) prefix
    - An optional equals sign (`=`) and default value, which must match the data type, if one was specified.
  - An optional trailing comma after the last parameter.
  - 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.

For example, this class definition specifies no parameters:

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

This class definition creates a version parameter (`$version`) that accepts a String data type with a default value of 'latest'. It also includes file content from an embedded Ruby (ERB) template from the `apache` module.

```
class apache (String $version = 'latest') {
  package {'httpd':
    ensure => $version, # Using the version parameter from above
    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'],
  }
}
```

### Class parameters and variables

Parameters allow a class to request external data. If a class needs to use data other than facts for configuration, you should usually use a parameter for that data.

You can use class parameters as normal variables inside the class definition. The values of these variables are set based on user input when the class is declared, rather than with normal assignment statements.

Supply default values for parameters whenever possible. If a class parameter lacks a default value, the parameter is considered required and the user must set a value, either in external data or as an override.

If you set a data type for each parameter, Puppet checks the parameter's value at runtime to make sure that it is the correct data type, and raises an error if the value is illegal. If you do not provide a data type for a parameter, the parameter accepts values of any data type.

The variables `$title` and `$name` are both set to the class name automatically, so you can't use them as parameters.

### Setting class parameter defaults with Hiera data

To set class parameter defaults with Hiera data in your modules, set up a hierarchy in your module's `hiera.yaml` file and include the referenced data files in the data directory.

For example, this `hiera.yaml` file, located in the root directory of the `ntp` module, uses the operating system fact to determine which class defaults to apply to the target system. Puppet first looks for a data file that matches the operating system of the target system: `path: "os/%{facts.os.family}.yaml"`. If no matching path is found, Puppet uses defaults from the common data file instead.

```
# ntp/hiera.yaml
---
version: 5
defaults:
  datadir: data
  data_hash: yaml_data
hierarchy:
  - name: "OS family"
    path: "os/%{facts.os.family}.yaml"

  - name: "common"
    path: "common.yaml"
```

The files in the example below specify the default values are located in the `data` directory:

- `AIX.yaml` specifies the defaults for systems that return an operating system fact of AIX.
- `Debian.yaml` specifies the defaults for systems that return an operating system fact of Debian.
- `common.yaml` specifies the defaults for all other systems.

```
# ntp/data/common.yaml
---
ntp::autoupdate: false
ntp::service_name: ntpd

# ntp/data/os/AIX.yaml
---
ntp::service_name: xntpd

# ntp/data/os/Debian.yaml
ntp::service_name: ntp
```

**Tip:**

If you are maintaining older modules, you might encounter cases where class parameter defaults are set with a parameter class, such as `params.pp`, and class inheritance. Update such modules to use Hiera data instead. Class inheritance can have unpredictable effects and makes troubleshooting difficult. For details about updating existing params classes to Hiera data, see data in modules.

### Related information

Variables on page 204
Variables store values so that those values can be accessed in code later.

Resources on page 207

Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.

Tags on page 364
Tags are useful for collecting resources, analyzing reports, and restricting catalog runs. Resources, classes, and defined type instances can have multiple tags associated with them, and they receive some tags automatically.

Scope on page 369
A scope is a specific area of code that is partially isolated from other areas of code.

Values and data types on page 294
Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

Namespaces and autoloading on page 374
Class and defined type names can be broken up into segments called namespaces which enable the autoloader to find the class or defined type in your modules.

## Declaring classes

Declaring a class in a Puppet manifest adds all of its resources to the catalog.

You can declare classes in node definitions, at top scope in the site manifest, and in other classes or defined types. Classes are singletons — although a given class can behave very differently depending on how its parameters are set, the resources in it are evaluated only once per compilation. You can also assign classes to nodes with an external node classifier (ENC) .

Puppet has two main ways to declare classes: include-like and resource-like. Include-like declarations are the most common; they are flexible and idempotent, so you can safely repeat them without causing errors. Resource-like declarations are mostly useful if you want to pass parameters to the class but can't or don't use Hiera. Most ENCs assign classes with include-like behavior, but others assign them with resource-like behavior. See the ENC interface documentation or the documentation of your specific ENC for details.

**CAUTION:** Do not mix include-like and resource-like declarations for a given class. If you declare or assign a class using both styles, it can cause compilation failures.

### Include-like declarations

Include-like resource declarations allow you to declare a class multiple times — but no matter how many times you add the class, it is added to the catalog only once. This allows classes or defined types to manage their own dependencies and allows you create overlapping *role* classes, in which a given node can have more than one role.

Include-like behavior relies on external data and defaults for class parameter values, which allows the external data source to act like cascading configuration files for all of your classes.

You can declare a class with this behavior with one of four functions: `include`, `require`, `contain`, and `hiera_include`.

When a class is declared with an include-like declaration, Puppet takes the following actions, in order, for each of the class parameters:

1. Requests a value from the external data source, using the key `<class name>::<parameter name>`. For example, to get the `apache` class's `version` parameter, Puppet searches for `apache::version`.
2. Uses the default value, if one exists.
3. Fails compilation with an error, if no value is found.

### The `include` function

The `include` function is the most common way to declare classes. Declaring a class with this function includes the class in the catalog.

**Tip:** The `include` function refers only to inclusion in the catalog. You can include a class in another class's definition, but doing so does not mean one class contains the other; it only means the included class will be added to the catalog. If you want one class to contain another, use the `contain` function instead.

This function uses include-like behavior, so you can make multiple declarations and Puppet relies on external data for parameters.

The `include` function accepts one of the following:

- A single class name, such as `apache`.
- A single class reference, such as `Class['apache']`.
- A comma-separated list of class names or class references.
- An array of class names or class references.

This single class name declaration declares the class only once and has no additional effect:

```
include base::linux
```

This example declare a single class with a class reference:

```
include Class['base::linux']
```

This example declares two classes in a list:

```

include base::linux, apache
```

This example declares two classes in an array:

```

$my_classes = ['base::linux', 'apache']
include $my_classes
```

#### The `require` function

The `require` function declares one or more classes, then causes them to become a dependency of the surrounding container. This function uses include-like behavior, so you can make multiple declarations, and Puppet relies on external data for parameters.

**Tip:** The `require` function is used to declare classes and defined types, and should not be confused with the `require` metaparameter, which is used to establish relationships between resources.

The `require` function accepts one of the following:

- A single class name, such as `apache`.
- A single class reference, such as `Class['apache']`.
- A comma-separated list of class names or class references.
- An array of class names or class references.

In this example, Puppet ensures that every resource in the `apache` class is applied before any resource in any `apache::vhost` instance:

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

### The `contain` function

The `contain` function is used inside another class definition to declare one or more classes and *contain* those classes in the surrounding class. This enforces ordering of classes. When you contain a class in another class, the relationships of the containing class extend to the contained class as well. For details about containment, see the documentation on containing classes.

This function uses include-like behavior, so you can make multiple declarations, and Puppet relies on external data for parameters.

The `contain` function accepts one of the following:

- A single class name, such as `apache`.
- A single class reference, such as `Class['apache']`.
- A comma-separated list of class names or class references.
- An array of class names or class references.

In this example class declaration, the `ntp` class contains the `ntp::service` class. Any resource that forms a relationship with the `ntp` class also has the same relationship to the `ntp::service` class.

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

For example, if a resource has a `before` relationship with the `ntp` class, that resource will also be applied before the `ntp::service` class. Similarly, any resource that forms a `require` relationship with `ntp` will be applied after `ntp::service`.

### The `hiera_include` function

The `hiera_include` function requests a list of class names from Hiera, then declares all of them.

This function uses include-like behavior, so you can make multiple declarations, and Puppet relies on external data for parameters. The `hiera_contain` function accepts a single lookup key.

Because `hiera_include` uses the array lookup type, it gets a combined list that includes classes from every level of the hierarchy. This allows you to abandon node definitions and use Hiera like a lightweight external node classifier. For more information, see the Hiera documentation.

For example, this `hiera_include` declaration in the site manifest applies classes across the site infrastructure, as specified in Hiera.

```
# /etc/puppetlabs/code/environments/production/manifests/site.pp
hiera_include(classes)
```

Given the Hiera data below, the node `web01.example.com` in the production environment gets the classes `apache`, `memcached`, `wordpress`, and `base::linux`. On all other nodes, only the `base::linux` class is declared.

```
# /etc/puppetlabs/puppet/hiera.yaml
...
hierarchy:
  - "%{::clientcert}"
  - common
```

```
# /etc/puppetlabs/code/hieradata/web01.example.com.yaml
---
classes:
  - apache
  - memcached
  - wordpress

# /etc/puppetlabs/code/hieradata/common.yaml
---
classes:
  - base::linux
```

#### Resource-like declarations

Resource-like class declarations require that you declare a given class only once. They allow you to override class parameters at compile time — for any parameters you don't override, Puppet falls back to external data.

Resource-like declarations must be unique to avoid conflicting parameter values. Repeated overrides cause catalog compilation to be unreliable and dependent on order evaluation. This is because overridden values from the class declaration:

- Always take precedence.
- Are computed at compile time.
- Do not have a built-in hierarchy for resolving conflicts.

When a class is declared with a resource-like declaration, Puppet takes the following actions, in order, for each of the class parameters:

1. Uses the override value from the declaration, if present.
2. Requests a value from the external data source, using the key `<class name>::<parameter name>`. For example, to get the `apache` class's `version` parameter, Puppet searches for `apache::version`.
3. Uses the default value.
4. Fails compilation with an error, if no value is found.

Resource-like declarations look like normal resource declarations, using the `class` pseudo-resource type. You can provide a value for any class parameter by specifying it as a resource attribute.

You can also specify a value for any metaparameter. In such cases, every resource contained in the class will also have that metaparameter. However:

- Any resource can specifically override metaparameter values received from its container.
- Metaparameters that can take more than one value, such as the relationships metaparameters, merge the values from the container and any resource-specific values.
- You cannot apply the `noop` metaparameter to resource-like class declarations.

For example, this resource-like declaration declares a class with no parameters:

```
class {'base::linux':}
```

This declaration declares a class and specifies the version parameter:

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

#### Related information

Node definitions on page 252

A node definition, also known as a node statement, is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.

Main manifest directory on page 85
Puppet starts compiling a catalog either with a single manifest file or with a directory of manifests that are treated like a single file. This starting point is called the *main manifest* or *site manifest*.

Defined resource types on page 227
*Defined resource types*, sometimes called defined types or defines, are blocks of Puppet code that can be evaluated multiple times with different parameters.

Relationships and ordering on page 214
Puppet normally applies resources in the order they are declared in their manifest. To manage a group of resources in a specific order, explicitly declare such relationships with relationship metaparameters, chaining arrows, and the `require` function.

Containment of resources on page 367
Containment enables you to control where and when specific parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types.

Resources on page 207
Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.

Writing external node classifiers on page 481
An external node classifier (ENC) is a script or application that tells Puppet which classes a node should have. It can replace or work in concert with the node definitions in the main site manifest (`site.pp`).

## Defined resource types

*Defined resource types*, sometimes called defined types or defines, are blocks of Puppet code that can be evaluated multiple times with different parameters.

Create a defined resource type by writing a `define` statement in a manifest (`.pp`) file. You can declare a resource of a defined type in the same way you would declare a resource of a built-in type.

Store defined resource type manifests in the `manifests/` directory of a module. Define only one defined type in a manifest, and give the manifest file the same name as the defined type. Puppet automatically loads any defined types that are present in a valid module. See module fundamentals to learn more about module structure and usage.

If a defined type is present and loadable, you can declare resources of that defined type anywhere in your manifests. Declaring a new resource of the defined type causes Puppet to re-evaluate the block of code in the definition, using different values for the parameters.

Every instance of a defined type contains all of its unique resources. This means that any relationships formed between the instance and another resource are extended to every resource that makes up the instance. See the topics about containment and relationships for more information.

**Tip:** Unlike many parts of Puppet code, define statements aren't expressions, so you can't use them where a value is expected.

### Defining types

The general form of a define statement is:

- The `define` keyword.
- The name of the defined type.

- An optional parameter list, which consists of:
  - An opening parenthesis.
  - A comma-separated list of parameters, such as: `String $myparam = "default value"`. Each parameter consists of:
    - An optional data type, which restricts the allowed values for the parameter. If no data type is specified, values of any data type are permitted.
    - A variable name to represent the parameter, including the `$` prefix, such as `$parameter`.
    - An optional equals = sign and default value, which must match the data type, if one was specified. If no default value is specified, the parameter is considered required and the user must specify a value.
  - An optional trailing comma after the last parameter.
  - 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 add the code to the catalog, you must declare one or more resources of the defined type.

This example creates a new resource type called `apache::vhost`:

```
# /etc/puppetlabs/puppet/modules/apache/manifests/vhost.pp
define apache::vhost (
  Integer $port,
  String[1] $docroot,
  String[1] $servername = $title,
  String $vhost_name = '*',
) {
  include apache # contains package['httpd'] and service['httpd']
  include apache::params # contains common config settings

  $vhost_dir = $apache::params::vhost_dir

  # the template used below can access all of the parameters and variable
 from above.
  file { "${vhost_dir}/${servername}.conf":
    ensure  => file,
    owner   => 'www',
    group   => 'www',
    mode    => '0644',
    content => template('apache/vhost-default.conf.erb'),
    require  => Package['httpd'],
    notify    => Service['httpd'],
  }
}
```

### Declaring defined type resources

You can declare instances of a defined type—usually just called *resources*—the same way you declare any other resource: with a resource type, a title, and a set of attribute-value pairs. The parameters you added when defining the type, such as `$port`, become resource attributes, such as `port`, when you declare resources of the defined type.

Parameters that have a default value are considered optional parameters: if you don't specify them in the resource declaration, the default value is used. Parameters without defaults are required parameters, and you must specify a value for them when you declare the resource.

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

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

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

If a defined type is present and loadable, you can declare resources of that defined type anywhere in your manifests. Declaring a new resource of the defined type causes Puppet to re-evaluate the block of code in the definition, using the new declaration's values for the parameters.

Just as with a normal resource type, you can declare resource defaults for a defined type. In this example, every `apache::vhost` resource defaults to port 80 unless specifically overridden:

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

You can include any metaparameter in the declaration of a defined type instance. If you do:

- Every resource contained in the resource declaration also has that metaparameter. Metaparameters that can accept more than one value, such as the relationship metaparameters, 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']`.

### Naming

Defined type names can consist of one or more *namespace* segments, which indicate the defined type's location in a module. Each segment must adhere to the naming and reserved names guidelines.

Each namespace segment must be capitalized when writing a resource reference, collector, or resource default. For example, a reference to the `apache::vhost` resource would be `Apache::Vhost['homepages']`.

Because you can declare multiple instances of a defined type in your manifests, every resource in the definition must be different in every instance. Duplicate resource instances result in compilation failures with a "duplicate resource declaration" error. To make resources different across instances, include the value of `$title` or another parameter in the resource's title and name.

Because `$title` is unique per instance, this ensures the resources are unique as well. For example, this segment of a file declaration makes resources unique by adding the `vhost_dir` and `servername` attributes to the resource title:

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

### Parameters and attributes

When you create a defined type, you can precede each parameter in the define statement with an optional data type. If you include a data type, Puppet checks the resource parameter's value at runtime to make sure that it has the right data type; if the value is illegal, Puppet raises an error. If you don't specify a data type in the definition statement, the parameter accepts values of any data type.

You can use the parameters of a defined type as local variables inside the definition. Rather than the usual assignment statement, each instance of the defined type uses its parameter attributes to set the value of the variable. In this example declaration, the value of the `port` parameter, 8081, becomes the value assigned to the `$port` variable. Likewise, the path for the `docroot` parameter becomes the value for the `$docroot` variable.

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

**Note:**

The `$title` and `$name` variables are both set to the defined type's name automatically, so they cannot be used as parameters.

### `$title` and `$name`

The `$title` and `$name` attributes are always available to a defined type and are not explicitly added to the definition. These attributes are both set to the defined type's name automatically:

- `$title` is always set to the title of the instance. Because it is always unique for each instance, it is useful for making sure that contained resources are unique.
- `$name` defaults to the value of `$title`. You can specify a different value when you declare an instance of the defined type, but this is rarely useful.

Because the values of `$title` and `$name` are already available inside the defined type's parameter list, you can use `$title` as all or part of the default value for another attribute. In this example, `$title` is used as the value of `$servername` to ensure the server name is always unique:

```
define apache::vhost (
  Integer $port,
  String[1] $docroot,
  String $servername = $title,
  String[1] $vhost_name = '*',
) { # ...
```

**Related information**

Resources on page 207
Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.

Resource default statements on page 358
Resource default statements enable you to set default attribute values for a given resource type. Resource declarations within the area of effect that omits those attributes inherit the default values.

Containment of resources on page 367
Containment enables you to control where and when specific parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types.

Namespaces and autoloading on page 374
Class and defined type names can be broken up into segments called namespaces which enable the autoloader to find the class or defined type in your modules.

Values and data types on page 294
Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

## Bolt tasks

Bolt tasks are single actions that you can run on target nodes in your infrastructure, allowing you to make as-needed changes to remote systems. You can run tasks with the Puppet Enterprise (PE) orchestrator or with Puppet's standalone task runner, Bolt.

Sometimes you need to do arbitrary tasks in your infrastructure that aren't about enforcing the state of machines. You might need to restart a service, run a troubleshooting script, or get a list of the network connections to a given node.

Tasks allow you to do actions like these with either the PE orchestrator or with Bolt, a standalone task runner. The orchestrator uses PE's built-in communication protocol, SSH, or WinRM to connect to the targets. Bolt connects to the targets with SSH or WinRM, without requiring any existing Puppet installation on the target. Bolt can also run plans, which chain multiple tasks together for more complex actions.

You can write tasks, which are a lot like scripts, in any programming language that can run on the target nodes, such as Bash, Python, or Ruby. Tasks are packaged within modules, so you can reuse, download, and share tasks on the Forge. Metadata for each task describes the task, validates input, and controls how the task runner executes the task.

For more information, see the Bolt documentation, particularly Tasks and plans. If you're a Puppet Enterprise user, see Running tasks and Using Bolt with orchestrator.

## Type aliases

Type aliases allow you to create reusable and descriptive data types and resource types.

By using type aliases, you can:

- Give a type a descriptive name, such as `IPv6Addr`, instead of creating or using a complex pattern-based type.
- Shorten and move complex type expressions.
- Improve code quality by reusing existing types instead of inventing new types.
- Test type definitions separately from manifests.

Type aliases are transparent, which means they are fully equivalent to the types of which they are aliases. For example, in the following code, the `notice` returns `true` because `MyType` is an alias of the `Integer` type:

```
type MyModule::MyType = Integer
notice MyModule::MyType == Integer
```

**Note:** The internal types `TypeReference` and `TypeAlias` are never values in Puppet code .

### Creating type aliases

Use the following syntax to create a type alias:

```
type <MODULE NAME>::<ALIAS NAME> = <TYPE DEFINITION>
```

The `<MODULE NAME>` must be named after the module that contains the type alias, and both the `<MODULE NAME>` and `<ALIAS NAME>` begin with a capital letter and must not be a reserved word.

For example, you can create a type alias named `MyType` that is equivalent to the `Integer` data type:

```
type MyModule::MyType = Integer
```

You can then declare a parameter using the alias as though it were a unique data type:

```
MyModule::MyType $example = 10
```

To make your code easier to maintain and troubleshoot, store type aliases as `.pp` files in your module's `types` directory, which is a top-level directory and sibling of the `manifests` and `lib` directories. Define only one alias per file, and name the file after the type alias name converted to lowercase. For example, `MyType` is expected to be loaded from a file named `mytype.pp`.

You can create recursive type aliases, which can refer to the alias being declared or to other types, thereby defining complex, descriptive type definitions without using the `Any` type. For example:

```
type MyModule::Tree = Array[Variant[Data, Tree]]
```

This `Tree` type alias is defined as a being built out of Arrays that contain Data, or a Tree:

```
[1,2 [3], [4, [5, 6], [[[[1,2,3]]]]]]
```

You can also create aliases to resource types:

```
type MyModule::MyFile = File
```

When defining an alias to a resource type, use its short form (for example, `File`) instead of its long form (`Resource[File]`).

#### Related information

Data type syntax on page 326
Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Resources on page 207
Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.

## Expressions and operators

*Expressions* are statements that resolve to values. You can use expressions almost anywhere a value is required. Expressions can be compounded with other expressions, and the entire combined expression resolves to a single value.

In the Puppet language, nearly everything is an expression, including literal values, references to variables, resource declarations, function calls, and more. In other words, almost all statements in the language resolve to a value and can be used anywhere that value would be expected.

Most of this page is about expressions that are constructed with *operators*. Operators take input values and operate on them (for example, mathematically) to result in some other value. Other kinds of expressions (for example, function calls) are described in more detail on other pages.

Some expressions have side effects and are used in Puppet primarily for their side effects, rather than for their result value. For example:

- Resource declaration: Adds a resource to the catalog.
- Variable assignment: Creates a variable and assigns it a value.
- Chaining statement: Forms a relationship between two or more resources.

Your code won't usually do anything with the value these expressions produce, but sometimes the value is useful for things like forming relationships to resources whose names can't be predicted until run time.

**Important:** The following statements are not typical expressions. They don't resolve to usable values and can only be used in certain contexts:

- Class definitions
- Defined types
- Node definitions
- Resource collectors
- Lambdas

Expressions can be used almost everywhere, including:

- The operand of another expression.
- The condition of an if statement.
- The control expression of a case statement or selector statement.
- The assignment value of a variable.
- The argument or arguments of a function call.
- The title of a resource.

- An entry in an array
- A key or value of a hash.

Expressions cannot be used:

- Where a literal name of a class or defined type is expected (for example, in `class` or `define` statements).
- As the name of a variable (the name of the variable must be a literal name).
- Where a literal resource type or name of a resource type is expected (for example, in the type position of a resource declaration).

You can surround an expression by parentheses to control the order of evaluation in compound expressions (for example, `10+10/5` is 12, and `(10+10)/5` is 4), or to make your code clearer.

For formal descriptions of expressions constructed with operators and other elements of the Puppet language, see the Puppet language specification.

### Operator expressions

There are two kinds of operators:

- *Infix operators*, also called *binary operators*, appear between two operands:
  - `$a = 1`
  - `5 < 9`
  - `$operatingsystem != 'Solaris'`
- *Prefix operators*, also called *unary operators*, appear immediately before a single operand:
  - `*$interfaces`
  - `!$is_virtual`

Operands in an expression can be any other expression — anything that resolves to a value of the expected data type is allowed. Each operator has its own rules, described in the sections below, for the data types of its operands.

When you create compound expressions by using other expressions as operands, use parentheses for clarity and readability:

```
(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. Expressions wrapped in parentheses are evaluated first, starting from the innermost expression:

```
# This example resolves to 30, not 23:
notice( (7+8)*2 )
```

For the sake of clarity, use parentheses in all but the simplest compound expressions.

The precedence of operators, from highest to lowest, is:

| Precedence | Operator |
| --- | --- |
| 1 | `!` (unary: not) |
| 2 | `-` (unary: numeric negation) |
| 3 | `*` (unary: array splat) |
| 4 | `in` |

| Precedence | Operator |
| --- | --- |
| 5 | `=~` and `!~` (regex or data type match or non-match) |
| 6 | `*`, `/`, `%` (multiplication, division, and modulo) |
| 7 | `+` and `-` (addition/ array concatenation and subtraction/array deletion) |
| 8 | `<<` and `>>` (left shift and right shift) |
| 9 | `==` and `!=` (equal and not equal) |
| 10 | `>=`, `<=`, `>`, and `<` (greater or equal, less or equal, greater than, and less than) |
| 11 | `and` |
| 12 | `or` |
| 13 | `=` (assignment) |

**Related information**

Values and data types on page 294
Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

Variables on page 204
Variables store values so that those values can be accessed in code later.

Resources on page 207
Resources are the fundamental unit for modeling system configurations. Each resource describes the desired state for some aspect of a system, like a specific service or package. When Puppet applies a catalog to the target system, it manages every resource in the catalog, ensuring the the actual state matches the desired state.

Function calls on page 249
Functions are plug-ins, written in Ruby, that you can call during catalog compilation. A call to any function is an expression that resolves to a value. Most functions accept one or more values as arguments, and return a resulting value.

Conditional statements and expressions on page 242
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. Puppet supports *if* and *unless* statements, *case* statements, and *selectors*.

## Comparison operators

Comparison operators take operands of several data types, and resolve to Boolean values.

Comparisons of numbers convert the operands to and from floating point and integer values, such that `1.0 == 1` is true. However, keep in mind that floating point values created by division are inexact, so mathematically equal values can be slightly unequal when turned into floating point values.

You can compare any two values with equals `==` or not equals `!=`, but only strings, numbers, and data types that require values to have a defined order can be compared with the less than or greater than operators.

**Note:** Comparisons of string values are case insensitive for characters in the US ASCII range. Characters outside this range are case sensitive.

Characters are compared based on their encoding. For characters in the US ASCII range, punctuation comes before digits, digits are in the order 0, 1, 2, ... 9, and letters are in alphabetical order. For characters outside US ASCII, ordering is defined by their UTF-8 character code, which might not always place them in alphabetical order for a given locale.

#### == (equality)

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

- Numbers: Tests simple equality.
- Strings: Tests whether two strings are identical, ignoring case as described in the Note, above.
- Arrays and hashes: Tests whether two arrays or hashes are identical.
- Booleans: Tests whether two Booleans are the same value.
- Data types: Tests whether two data types would match the exact same set of values.

Values are considered equal only if they have the same data type. Notably, this means that `1 == "1"` is false, and `"true" == true` is false.

#### != (non-equality)

Resolves to `false` if the operands are equal. So, `$x != $y` is the same as `!($x == $y)`. It has the same behavior and restrictions, but opposite result, as equality ==, above.

#### < (less than)

Resolves to `true` if the left operand is smaller than the right operand. Accepts numbers, strings, and data types; both operands must be the same type. When acting on data types, a less-than comparison is `true` if the left operand is a subset of the right operand.

#### > (greater than)

Resolves to `true` if the left operand is larger than the right operand. Accepts numbers, strings, and data types; both operands must be the same type. When acting on data types, a greater-than comparison is `true` if the left operand is a superset of the right operand.

#### <= (less than or equal to)

Resolves to `true` if the left operand is smaller than or equal to the right operand. Accepts numbers, strings, and data types; both operands must be the same type. When acting on data types, a less-than-or-equal-to comparison is `true` if the left operand is the same as the right operand or is a subset of it.

#### >= (greater than or equal to)

Resolves to `true` if the left operand is larger than or equal to the right operand. Accepts numbers, strings, and data types; both operands must be the same type. When acting on data types, a greater-than-or-equal-to comparison is `true` if the left operand is the same as the right operand or is a superset of it.

#### =~ (regex or data type match)

Resolves to `true` if the left operand matches the right operand. *Matching* means different things, depending on what the right operand is.

This operator is non-transitive with regard to data types. The right operand must be one of:

- A regular expression (regex), such as `/^[<>=]{7}/`.
- A stringified regular expression — that is, a string that represents a regular expression, such as `"^[<>=]{7}"`.
- A data type, such as `Integer[1,10]`.

If the right operand is a regular expression or a stringified regular expression, the left operand must be a string, and the expression resolves to `true` if the string matches the regular expression.

If the right operand is a data type, the left operand can be any value. The expression resolves to `true` if the left operand has the specified data type. For example, `5 =~ Integer` and `5 =~ Integer[1,10]` are both `true`.

#### `!~` (regex or data type non-match)

Resolves to `false` if the left operand matches the right operand. So, `$x !~ $y` is the same as `!($x =~ $y)`. It has the same behavior and restrictions, but opposite result, as regex match `=~`, above.

#### `in`

Resolves to `true` if the right operand contains the left operand. The exact definition of "contains" here depends on the data type of the right operand. See table below.

This operator is non-transitive with regard to data types. It accepts:

- A string, regular expression, or data type as the left operand.
- A string, array, or hash as the right operand.

| Expression | How `in` expression is evaluated |
| --- | --- |
| String `in` String | Tests whether the left operand is a substring of the right, ignoring case: `'eat' in 'eaten' # resolves to true` `'Eat' in 'eaten' # resolves to true` |
| String `in` Array | Tests whether one of the members of the array is identical to the left operand, ignoring case: `'eat' in ['eat', 'ate', 'eating'] # resolves to true` `'Eat' in ['eat', 'ate', 'eating'] # resolves to true` |
| String `in` Hash | Tests whether the hash has a *key* identical to the left operand, ignoring case: `'eat' in { 'eat' => 'present tense', 'ate' => 'past tense'} # resolves to true` `'eat' in { 'present' => 'eat', 'past' => 'ate' } # resolves to false` |

<table>
<tr><th>Expression</th><th>How <code>in</code> expression is evaluated</th></tr>
<tr><td>Regex <code>in</code> String</td><td>Tests whether the right operand matches the regular expression:
<pre>
# note the case-insensitive option ?i
/(?i:EAT)/ in 'eatery'
 # resolves to true
</pre></td></tr>
<tr><td>Regex <code>in</code> Array</td><td>Tests whether one of the members of the array matches the regular expression:
<pre>
/(?i:EAT)/ in ['eat',
 'ate', 'eating'] #
 resolves to true
</pre></td></tr>
<tr><td>Regex <code>in</code> Hash</td><td>Tests whether the hash has a key that matches the regular expression:
<pre>
/(?i:EAT)/ in { 'eat'
 => 'present tense',
 'ate' => 'past tense'}
 # resolves to true
/(?i:EAT)/ in
 { 'present' => 'eat',
 'past' => 'ate' } #
 resolves to false
</pre></td></tr>
<tr><td>Data type <code>in</code> Array</td><td>Tests whether one of the members of the array matches the data type:
<pre>
# looking for integers
 between 100 and 199
Integer[100, 199] in
 [1, 2, 125] # resolves
 to true
Integer[100, 199] in
 [1, 2, 25]  # resolves
 to false
</pre></td></tr>
<tr><td>Data type <code>in</code> anything else</td><td>Always <code>false</code>.</td></tr>
</table>

**Related information**

Booleans on page 307
Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.

Numbers on page 304
Numbers in the Puppet language are normal integers and floating point numbers.

Strings on page 295
Strings are unstructured text fragments of any length. They're a common and useful data type.

Arrays on page 308

Arrays are ordered lists of values. Resource attributes which accept multiple values (including the relationship metaparameters) generally expect those values in an array. Many functions also take arrays, including the iteration functions.

Hashes on page 311
Hashes map keys to values, maintaining the order of the entries according to insertion order.

Data type syntax on page 326
Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Regular expressions on page 314
A regular expression (sometimes shortened to "regex" or "regexp") is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

## Boolean operators

Boolean expressions resolve to boolean values. They are most useful when creating compound expressions.

A boolean operator takes boolean operands. If you pass in another type, it will be converted to boolean; see the section Automatic conversion to boolean in the data type documentation.

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

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

**Related information**

Booleans on page 307
Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.

## Arithmetic operators

Arithmetic expressions resolve to numeric values. Except for the unary negative `-`, arithmetic operators take two numeric operands. If an operand is a string, it's converted to numeric form. The operation fails if a string can't be converted.

### + (addition)

Resolves to the sum of the two operands.

### `-` (subtraction and negation)

When used with two operands, resolves to the difference of the two operands, left minus right. When used with one operand, returns the value of subtracting that operand from zero.

### `/` (division)

Resolves to the quotient of the two operands, the left divided by the right.

#### * (multiplication)

Resolves to the product of the two operands. The asterisk is also used as a unary splat operator for arrays (see below).

#### % (modulo)

Resolves to the remainder of dividing the left operand by the right operand:

```
5 % 2   # resolves to 1

32 % 7  # resolves to 4
```

#### << (left shift)

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

```
4 << 3   # resolves to 32: 4 times two cubed
```

#### >> (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:

```
16 << 3   # resolves to 2: 16 divided by two cubed
```

**Related information**

Numbers on page 304

Numbers in the Puppet language are normal integers and floating point numbers.

## Array operators

Array operators take arrays as operands, and, with the exception of `*` (unary splat), they resolve to array values.

#### * (splat)

The unary splat operator `*` accepts a single array value. If you pass it a scalar value, it converts the value to a single-element array first. The splat operator "unfolds" an array, resolving to a comma-separated list values representing the array elements. It's useful in places where a comma-separated list of values is valid, including:

- The arguments of a function call.
- The cases of a case statement.
- The cases of a selector statement.

If you use it in other contexts, it resolves to the array that was passed in.

For example:

```
$a = ['vim', 'emacs']
myfunc($a)    # Calls myfunc with a single argument, the array containing
 'vim' and 'emacs'
:
                   # myfunc(['vim','emacs'])
myfunc(*$a)   # Calls myfunc with two arguments, 'vim' and 'emacs':
                   # myfunc('vim','emacs')
```

Another example:

```
$a = ['vim', 'emacs']
$x = 'vim'
notice case $x {
  $a      : { 'an array with both vim and emacs' }
  *$a     : { 'vim or emacs' }
  default : { 'no match' }
}
```

### << (append)

Resolves to an array containing the elements in the left operand, with the right operand as its final element.

The left operand must be an array, and the right operand can be any data type. Appending adds only a single element to an array. To add multiple elements from one array to another, use the concatenation operator +.

Examples:

```
[1, 2, 3] << 4     # resolves to [1, 2, 3, 4]
[1, 2, 3] << [4, 5]   # resolves to [1, 2, 3, [4, 5]]
```

The append operator does not change its operands; it creates a new value.

### + (concatenation)

Resolves to an array containing the elements in the left operand followed by the elements in the right operand.

Both operands should be arrays. If the left operand isn't an array, Puppet interprets + as arithmetic addition. If the right operand is a scalar value, it is converted to a single-element array first.

Hash values are converted to arrays instead of wrapped, so you must wrap them yourself.

Examples:

```
[1, 2, 3] + 1     # resolves to [1, 2, 3, 1]
[1, 2, 3] + [1]   # resolves to [1, 2, 3, 1]
[1, 2, 3] + [[1]] # resolves to [1, 2, 3, [1]]
```

The concatenation operator does not change its operands; it creates a new value.

### - (removal)

Resolves to an array containing the elements in the left operand, with every occurrence of elements in the right operand removed.

Both operands should be arrays. If the left operand isn't an array, Puppet interprets - as arithmetic subtraction. If the right operand is a scalar value, it is converted to a single-element array first.

Hash values aren't automatically wrapped in arrays, so you must always wrap them yourself.

Examples:

```
[1, 2, 3, 4, 5, 1, 1] - 1    # resolves to [2, 3, 4, 5]
[1, 2, 3, 4, 5, 1, 1] - [1]  # resolves to [2, 3, 4, 5]
[1, 2, 3, [1, 2]] - [1, 2]   # resolves to [3, [1, 2]]
[1, 2, 3, [1, 2]] - [[1, 2]] # resolves to [1, 2, 3]
```

The removal operator does not change its operands; it creates a new value.

#### Related information

Arrays on page 308

Arrays are ordered lists of values. Resource attributes which accept multiple values (including the relationship metaparameters) generally expect those values in an array. Many functions also take arrays, including the iteration functions.

## Hash operators

Hash operators accept hashes as their left operand, and hashes or specific kinds of arrays as their right operand. The expressions resolve to hash values.

### + (merging)

Resolves to a hash containing the keys and values in the left operand and the keys and values in the right operand. If a key is present in both operands, the final hash uses the value from the right. It does not merge hashes recursively; it only merges top-level keys.

The right operand can be one of the following:

- A hash
- An array with an even number of elements. Each pair is converted in order to a key-value hash pair.

Examples:

```
{a => 10, b => 20} + {b => 30}  # resolves to {a => 10, b => 30}
{a => 10, b => 20} + {c => 30}  # resolves to {a => 10, b => 30, c => 30}
{a => 10, b => 20} + [c, 30]    # resolves to {a => 10, b => 20, c => 30}
{a => 10, b => 20} + 30         # gives an error
{a => 10, b => 20} + [30]       # gives an error
```

The merging operator does not change its operands; it creates a new value.

### - (removal)

Resolves to a hash containing the keys and values in the left operand, minus any keys that are also present in the right operand.

The right operand can be one of the following:

- A hash. The keys present in this hash will be absent in the final hash, regardless of whether that key has the same values in both operands. The key, not the value, determines whether it's removed.
- An array of keys.
- A single key.

Examples:

```
{a => first, b => second, c => 17} - {c => 17, a => "something else"} #
 resolves to {b => second}
{a => first, b => second, c => 17} - {a => a, d => d}                 #
 resolves to {b => second, c => 17}
{a => first, b => second, c => 17} - [c, a]                           #
 resolves to {b => second}
{a => first, b => second, c => 17} - c                                #
 resolves to {a => first, b => second}
```

The removal operator does not change its operands; it creates a new value.

#### Related information

Hashes on page 311

Hashes map keys to values, maintaining the order of the entries according to insertion order.

## Assignment operator

Puppet has one assignment operator, `=`.

### = (assignment)

The assignment operator sets the variable on the left side to the value on the right side. The expression resolves to the value of the right hand side. Variables can be set only one time, after which, attempts to set the variable to a new value cause an error.

#### Related information

Variables on page 204
Variables store values so that those values can be accessed in code later.

# Conditional statements and expressions

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. Puppet supports *if* and *unless* statements, *case* statements, and *selectors*.

### Examples

An `if` statement evaluates the given condition and, if the condition resolves to `true`, executes the given code. This example includes an `elsif` condition, and gives a warning if you try to include the `ntp` class on a virtual machine or on machine running macOS:

```
if $facts['is_virtual'] {
  warning('Tried to include class ntp on virtual machine; this node might be
 misclassified.')
}
elsif $facts['os']['family'] == 'Darwin' {
  warning('This NTP module does not yet work on our Mac laptops.')
}
else {
  include ntp
}
```

An `unless` statement takes a Boolean condition and an arbitrary block of Puppet code, evaluates the condition, and if the condition is false, execute the code block. This statement sets `$maxclient` to 500 unless the system memory is above the specified parameter.

```
unless $facts['memory']['system']['totalbytes'] > 1073741824 {
  $maxclient = 500
}
```

A case statement evaluates a list of cases against a control expression, and executes the first code block where the case value matches the control expression. This example declares a role class on a node, but which role class it declares depends on what operating system the node runs:

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

A selector statement is similar to a case statement, but instead of executing code, it returns a value. This example returns the value 'wheel' for the specified operating systems, but the value 'root' for all other operating systems:

```
$rootgroup = $facts['os']['family'] ? {
  'Solaris'                    => 'wheel',
  /(Darwin|FreeBSD)/  => 'wheel',
  default                      => 'root',
}

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

## `if` statements

An *"i"f statement* takes a Boolean condition and an arbitrary block of Puppet code, and executes the code block only if the condition is true. Optionally, an `if` statement can include `elsif` and `else` clauses.

### Behavior

Puppet's `if` statements behave much like those in any other language. The `if` condition is evaluated first and, if it is true, 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 does nothing and moves on. If statements executes a maximum of one code block.

In addition to executing the code in a block, an `if` statement also produces a value, so the `if` statement can be used wherever a value is allowed.The value of an `if` expression is the value of the last expression in the executed block, or `undef` if no block was executed.

### Syntax

An `if` statement consists of:

- The `if` keyword.
- A condition (any expression resolving to a Boolean value).
- A pair of curly braces containing any Puppet code.
- Optionally: any number of `elsif` clauses, which are processed in order.
- Optionally: the `else` keyword and a pair of curly braces containing Puppet code.

An `elsif` clause consists of:

- The `elsif` keyword.
- A condition.
- A pair of curly braces containing any Puppet code.

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

### Conditions

The condition of an `if` statement can be any expression that resolves to a Boolean value. This includes:

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

Expressions that resolve to non-Boolean values are automatically converted to Booleans. For more information, see the Booleans documentation.

### Regex capture variables

If you use the regular expression match operator in a condition, any captures from parentheses in the pattern are available inside the associated code block as numbered variables (for example, `$1`, `$2`), and the entire match is available as `$0`. This example captures any digits from a hostname such as `www01` and `www02`, and stores them in the `$1` variable:

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

Regex capture variables are different from other variables in a couple of ways:

- 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.

**Related information**

Scope on page 369
A scope is a specific area of code that is partially isolated from other areas of code.

Booleans on page 307
Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.

## `unless` statements

*"Unles"s statements* work like reversed `if` statements. They take a Boolean condition and an arbitrary block of Puppet code, evaluate the condition, and if it is false, execute the code block. They cannot include `elsif` clauses.

### Behavior

The condition is evaluated first and, if it is false, the code block is executed. If the condition is true, Puppet does nothing and moves on.

In addition to executing the code in a block, an `unless` statement is also an expression that produces a value, and it can be used wherever a value is allowed. The value of an `unless` expression is the value of the last expression in the executed block. If no block was executed, the value is `undef`.

### Syntax

The general form of an `unless` statement is:

- The `unless` keyword.
- A condition (any expression resolving to a Boolean value).
- A pair of curly braces containing any Puppet code.
- Optionally: the `else` keyword and a pair of curly braces containing Puppet code.

You cannot include an `elsif` clause in an `unless` statement. If you do, compilation fails with a syntax error.

```
unless $facts['memory']['system']['totalbytes'] > 1073741824 {
  $maxclient = 500
}
```

#### Conditions

The condition of an `unless` statement can be any expression that resolves to a Boolean value. This includes:

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

Expressions that resolve to non-Boolean values are automatically converted to Booleans. For more information, see the Booleans documentation.

#### Regex capture variables

Although `unless` statements receive regex capture variables like `if` statements, you wouldn't usually use one, because the code in the statement is executed only if the condition doesn't match anything. It generally makes more sense to use an `if` statement.

### 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 execute the first block whose case value matches the control expression.

#### Behavior

Puppet compares the control expression to each of the cases, in the order they are listed (except for the top-most level default case, which always goes last). It executes the block of code associated with the first matching case, and ignores the remainder of the statement.Case statements execute a maximum of one code block. If none of the cases match, Puppet does nothing and moves on.

In addition to executing the code in a block, a `case` statement is also an expression that produces a value, and can be used wherever a value is allowed.The value of a `case` expression is the value of the last expression in the executed block. If no block was executed, the value is `undef`.

The control expression of a case statement can be any expression that resolves to a value. This includes:

- Variables.
- Expressions.
- Functions that return values.

#### Syntax

The general form of a case statement is:

- The `case` keyword.
- A control expression, which is any expression resolving to a value.
- An opening curly brace.

- Any number of possible matches, which consist of:
  - A case or comma-separated list of cases.
  - A colon.
  - A pair of curly braces containing any arbitrary Puppet code.
  - A closing curly brace case.

```
case $facts['os']['name'] {
  'Solaris':           { include role::solaris } # Apply the solaris class
  'RedHat', 'CentOS':  { include role::redhat  } # Apply the redhat class
  /^(Debian|Ubuntu)$/: { include role::debian  } # Apply the debian class
  default:             { include role::generic } # Apply the generic class
}
```

### Case matching

A case can be any expression that resolves to a value, for example, literal values, variables and function calls. You can use a comma-separated list of cases to associate multiple cases with the same block of code. To use values from a variable as cases, use the `*` splat operator to convert an array of values into a comma-separated list of values.

Depending on the data type of a case's value, Puppet uses one of following behaviors to test whether the case matches:

- Most data types, for example, strings and Booleans, are compared to the control value with the `==` equality operator, which is case-insensitive when comparing strings.
- Regular expressions are compared to the control value with the `=~` matching operator, which is case-sensitive. Regex cases only match strings.
- Data types, such as `Integer`, are compared to the control value with the `=~` matching operator. This tests whether the control value is an instance of that data type.
- Arrays are recursively compared to the control value. First, Puppet checks whether the control and array are the same length, then each corresponding element is compared using these same case matching rules.
- Hashes compare each key-value pair. To match, the control value and the case must have the same keys, and each corresponding value is compared using these same case matching rules.
- The special value `default` matches anything, and unless nested inside an array or hash, is always tested last regardless of its position in the list.

### Regex capture variables

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

```
case $trusted['hostname'] {
  /www(\d+)/: { notice("Welcome to web server number ${1}"); include
 role::web }
  default:   { include role::generic }
}
```

This example captures any digits from a hostname such as `www01` and `www02` and store them in the `$1` variable.

Regex capture variables are different from other variables in a couple of ways:

- 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.

### Best practices

Case statements should 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 mysterious 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 maintains your code.

### Related information

Values and data types on page 294
Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

Regular expressions on page 314
A regular expression (sometimes shortened to “regex” or “regexp”) is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

Scope lookup rules on page 373
The scope lookup rules determine when a local scope becomes the parent of another local scope.

## Selector expressions

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

### Behavior

The entire selector expression is treated as a single value.Puppet compares the control expression to each of the cases, in the order they are listed (except for the `default` case, which always goes last). When it finds a matching case, it treats that value as the value of the expression and ignore the remainder of the expression.If none of the cases match, Puppet fails compilation with an error.

The control expression of a selector can be any expression that resolves to a value. This includes:

- Variables.
- Expressions.
- Functions that return values

Selectors can be used wherever a value is expected. This includes:

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

**Tip:** For readability sake, you should use selectors only in variable assignments.

### Syntax

Selectors resemble a cross between a case statement and the ternary operator found in other languages. The general form of a selector is:

- A control expression, which is any expression resolving to a value.
- The `?` (question mark) keyword.
- An opening curly brace.

- Any number of possible matches, each of which consists of:
  - A case.
  - The `=>` (hash rocket) keyword.
  - A value, which can be any expression resolving to a value.
  - A trailing comma.
- A closing curly brace.

In this example, the value of `$rootgroup` is determined using the value of `$facts['os']['family']`:

```
$rootgroup = $facts['os']['family'] ? {
  'Solaris'                    => 'wheel',
  /(Darwin|FreeBSD)/ => 'wheel',
  default                     => 'root',
}

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

### Case matching

In selector statements, you cannot use lists of cases. If the control expression is a string and you need more than one case associated with a single value, use a regular expression. Otherwise, use a case statement instead of a selector, because case statements do allow lists of cases. For more information, see Case statements.

### Regex capture variables

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

```
puppet
$system = $facts['os']['name'] ? {
  /(RedHat|Debian)/ => "our system is ${1}",
  default           => "our system is unknown",
}
```

Regex capture variables are different from other variables in a couple of ways:

- 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.

### Related information

Values and data types on page 294
Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

Regular expressions on page 314
A regular expression (sometimes shortened to “regex” or “regexp”) is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

Scope on page 369

A scope is a specific area of code that is partially isolated from other areas of code.

## Function calls

Functions are plug-ins, written in Ruby, that you can call during catalog compilation. A call to any function is an expression that resolves to a value. Most functions accept one or more values as arguments, and return a resulting value.

The Ruby code in the function can do any number of things to produce the final value, including:

- Evaluate templates.
- Do mathematical calculations.
- Look up values from an external source.
- Cause side effects that modify the catalog.
- Evaluate a provided block of Puppet code, possibly using the function's arguments to modify that code or control how it runs.

Puppet includes several built-in functions. More functions are available in modules, such as `puppetlabs-stdlib`, on the Forge. You can also write custom functions and put them in your own modules.

An entire function call—including the name, arguments, and lambda—constitutes an expression. It resolves to a single value, and can be used anywhere a value of that type is accepted. A function call might also have an effect, such as adding a class to the catalog. You can also use function calls on their own, which causes their effects to occur while their value is ignored.

All functions run during catalog compilation, which means they can access code and data only from the Puppet master. To make changes to an agent node, you must use a resource; to collect data from an agent node, you must use a custom fact.

Each function defines how many arguments it takes, what data types it expects those arguments to be, what values it returns, and any effects it has. For details about any functions built into Puppet, see the function reference. For details about a function included in a module, see that module's documentation.

### Statement functions

Statement functions are a group of built-in functions that are used only for their effects, rather than for any values. Puppet recognizes only its built-in statements; it doesn't allow adding new statement functions as plugins. The major difference between statement functions and other functions is that you can omit parentheses when calling a statement function with at least one argument, such as `include apache`.

Statement functions return a value like any other function, but they always return a value of `undef`. The built-in statement functions are:

| | |
|---|---|
| **Catalog statements** | `include`: Includes the specified classes in a catalog.<br>`require`: Includes the specified classes in the catalog and adds them as a dependency of the current class or defined resource<br>`contain`: Includes the specified classes in the catalog and contains them in the current class.<br>`tag`: Adds the specified tag or tags to the containing class or defined resource. |
| **Logging statements** | `debug`: Logs message at the debug level.<br>`info`: Logs message at the info level.<br>`notice`: Logs message at the notice level.<br>`warning`: Logs message at the warning level |

| | |
|---|---|
| | `err`: Logs message at the error level. |
| **Failure statements** | `fail`: Logs the error message and terminates compilation. |

#### Related information

Custom functions on page 376
Puppet includes many built-in functions, and more are available in modules on the Forge. You can also write your own custom functions.

## Functions syntax

Like any expression, a function call can be used anywhere the value it returns would be allowed. Function calls can also stand on their own, to cause their side effects, but ignore their returned value.

There are two ways to call functions in the Puppet language: *prefix calls* as in `template("ntp/ntp.conf.erb")`, and *chained calls* as in `"ntp/ntp.conf.erb".template`. There's also a modified form of prefix call that can only be used with certain functions.

The two function call styles have exactly the same capabilities, so you can choose whichever one is more readable. In general:

- For functions that take many arguments, prefix calls are easier to read.
- For functions that take one normal argument and a lambda, chained calls are easier to read.
- For a series of functions where each takes the last one's result as its argument, chained calls are easier to read, especially if at least one of those functions accepts a lambda.

Most functions have short, one-word names. The modern function API also allows qualified function names like `mymodule::myfunction`. Functions must always be called with their full names; you can't shorten a qualified function name.

### Prefix function calls

You call a function in the prefix style by writing its name and providing a list of arguments in parentheses. The general form of a prefix function call is:

```
function_name(argument, argument, ...) |$parameter, $parameter, ...| { code
 block }
```

- The full name of the function, as an unquoted word.
- An opening parenthesis `(`. Parentheses are optional when you're calling a built-in statement function with at least one argument, as in `include apache`. They're mandatory in all other cases.
- Zero or more *arguments*, separated by commas. Arguments can be any expression that resolves to a value. See each function's docs for the number of its arguments and their data types. Use the splat array operator `*` to convert an array into a comma-separated list of arguments.
- A closing parenthesis `)` if an opening parenthesis was used.
- Optionally, a lambda (code block), if the function accepts one.

In the following example, `template`, `include`, and `each` are all functions. The `template` function is used for its return value, `include` adds a class to the catalog, and `each` runs a block of code several times with different values.

```
file {"/etc/ntp.conf":
  ensure  => file,
  content => template("ntp/ntp.conf.erb"), # function call; resolves to a
 string
}

include apache # function call; modifies catalog
```

```
$binaries = [
  "facter",
  "hiera",
  "mco",
  "puppet",
  "puppetserver",
]

# function call with lambda; runs block of code several times
each($binaries) |$binary| {
  file {"/usr/bin/$binary":
    ensure => link,
    target => "/opt/puppetlabs/bin/$binary",
  }
}
```

#### Chained function calls

Alternatively, you call a function in the chained style by writing its first argument, a period, and the name of the function. The general form of a chained function call is:

```
argument.function_name(argument, ...) |$parameter, $parameter, ...| { code
 block }
```

- The first argument of the function, which can be any expression that resolves to a value.
- A period `.`
- The full name of the function, as an unquoted word.
- Optionally, parentheses containing a comma-separated list of additional arguments, starting with the second argument (because you already passed in the first argument). Use the splat array operator `*` to convert an array to a comma-separated list of arguments.
- Optionally, a lambda (code block), if the function accepts one.

In the following example, `template`, `include`, and `each` are all functions. The `template` function is used for its return value, `include` adds a class to the catalog, and `each` runs a block of code several times with different values.

```
puppet
file {"/etc/ntp.conf":
  ensure  => file,
  content => "ntp/ntp.conf.erb".template, # function call; resolves to a
 string
}

apache.include # function call; modifies catalog

$binaries = [
  "facter",
  "hiera",
  "mco",
  "puppet",
  "puppetserver",
]

# function call with lambda; runs block of code several times
$binaries.each |$binary| {
  file {"/usr/bin/$binary":
    ensure => link,
    target => "/opt/puppetlabs/bin/$binary",
  }
}
```

#### Related information

Lambdas on page 355
Lambdas are blocks of Puppet code passed to functions. When a function receives a lambda, it provides values for the lambda's parameters and evaluates its code. If you use other programming languages, think of lambdas as anonymous functions that are passed to other functions.

Expressions and operators on page 232
*Expressions* are statements that resolve to values. You can use expressions almost anywhere a value is required. Expressions can be compounded with other expressions, and the entire combined expression resolves to a single value.

Values and data types on page 294
Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

Array operators on page 239
Array operators take arrays as operands, and, with the exception of `*` (unary splat), they resolve to array values.

## Node definitions

A node definition, also known as a node statement, is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.

Put node definitions in the main manifest, which can be a single `site.pp` file, or a directory containing many files.

If the main manifest contains at least one node definition, it must have one for *every* node. Compilation for a node fails if a node definition for it cannot be found. Either specify no node definitions, or use the `default` node definition, as described below, to avoid this situation.

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

Node definitions create an anonymous scope that can override variables and defaults from top scope.

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

Node definitions are an optional feature of Puppet. You can use them instead of or in combination with an external node classifier (ENC). Alternatively, you can use conditional statements with facts to classify nodes. Unlike more general conditional structures, node definitions match nodes only by name. By default, the name of a node is its certname, which defaults to the node's fully qualified domain name.

Although you can use node definitions in conjunction with an ENC, it's simpler to choose one method or the other. If you do use them together, Puppet merges their data as follows:

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

### Syntax

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

- The `node` keyword.
- The node definition name: a quoted string, a regular expression, or `default`.
- An opening curly brace.
- Any mixture of class declarations, variables, resource declarations, collectors, conditional statements, chaining relationships, and functions.
- A closing curly brace.

In the following example, only `www1.example.com` receives the `apache` and `squid` classes, and only `db1.example.com` receives the `mysql` class:

```
# <ENVIRONMENTS DIRECTORY>/<ENVIRONMENT>/manifests/site.pp
node 'www1.example.com' {
  include common
  include apache
  include squid
}
node 'db1.example.com' {
  include common
  include mysql
}
```

A node definition name must be one of the following:

- A quoted string containing only letters, numbers, underscores (_), hyphens (`-`), and periods (`.`).
- A regular expression.
- The bare word `default`. If no other node definition matches a given node, the `default` node definition will be used for that node.

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

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

If you use a regular expression for a node definition name, it also has the potential to match multiple nodes. For example, the following node definition matches `www1`, `www13`, and any other node whose name consists of `www` and one or more digits:

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

The following example of a regex node definition name matches `one.example.com` and `two.example.com`, but no other nodes:

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

**Important:** Make sure all of your node definition name regexes match non-overlapping sets of node names. If a node's name matches more than one regex, Puppet makes no guarantee about which matching definition it will get.

You can use regex capture variables by enclosing parts of your regex node definition name in parentheses `()`, and then referencing them in order as `$1`, `$2` and so on, as variables within the body of the node definition. For example:

```
node /^www(\d+)$/ {
  $wwwnumber = $1  #assigns the value of the (\d+) from a regex match to the
 variable $wwwnumber
}
```

### Matching

A given node gets the contents of only **one** node definition, even if multiple node definitions could match its name. Puppet does the following checks, in this order, until it finds one that matches:

1. If there is a node definition with the node's exact name, Puppet uses it.

2. If there is a regular expression node definition that matches the node's name, Puppet uses it. If more than one regex node matches, Puppet uses one of them, but we can't predict which. Make your node definition name regexes non-overlapping to avoid this problem.
3. If the node's name looks like a fully qualified domain name (it has multiple period-separated groups of letters, numbers, underscores, and dashes), Puppet chops off the final group and start again at step 1.

   You can turn off this fuzzy name checking by changing the master's `strict_hostname_checking` configuration setting to `true`. This causes Puppet to skip this looping step and use only the node's full name before resorting to `default`.
4. Puppet uses the `default` node.

For example, when compiling a catalog for a node with certname `www01.example.com`, with fuzzy checking, Puppet looks for a node definition with the following name, in this order:

- `www01.example.com`
- A regex that matches `www01.example.com`
- `www01.example`
- A regex that matches `www01.example`
- `www01`
- A regex that matches `www01`
- `default`

If it doesn't find one, catalog compilation fails. It's a good idea to always have a `default` node definition.

#### Related information

Variables on page 204
Variables store values so that those values can be accessed in code later.

Conditional statements and expressions on page 242
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. Puppet supports *if* and *unless* statements, *case* statements, and *selectors*.

Key configuration settings on page 62
Puppet has about 200 settings, all of which are listed in the configuration reference. Most of the time, you interact with only a couple dozen of them. This page lists the most important ones, assuming that you're okay with default values for things like the port Puppet uses for network traffic. See the configuration reference for more details on each.

Main manifest directory on page 85
Puppet starts compiling a catalog either with a single manifest file or with a directory of manifests that are treated like a single file. This starting point is called the *main manifest* or *site manifest*.

Scope on page 369
A scope is a specific area of code that is partially isolated from other areas of code.

## Facts and built-in variables

Before requesting a catalog for a managed node, or compiling one with `puppet apply`, Puppet collects system information, called *facts*, by using the Facter tool. The facts are assigned as values to variables that you can use anywhere in your manifests. Puppet also sets some additional special variables, called *built-in variables*, which behave a lot like facts.

Puppet code can access the following facts when compiling a catalog:

- Core facts from Facter.
- Custom facts and external facts that are present in your modules.

To see the fact values for a node, run `facter -p` on the command line, or browse facts on node detail pages in the Puppet Enterprise console. You can also use the PuppetDB API to explore or build tools to search and report on your infrastructure's facts.

Puppet honors fact values of of any data type. It does not convert Boolean, numeric, or structured facts to strings.

## Accessing facts from Puppet code

When you write Puppet code, you can access facts in two ways: with the `$fact_name` syntax, or with the `$facts['fact_name']` hash.

### Using the `$fact_name` syntax

Facts appear in Puppet as top-scope variables. They can be accessed in manifests as `$fact_name`.

For example:

```
if $osfamily == 'RedHat' {
  # ...
}
```

**Tip:** When you code with this fact syntax, it's not immediately obvious that you're using a fact — someone reading your code needs to know which facts exist to guess that you're accessing a top-scope variable. To make your code easier for others to read, use the `$::fact_name` syntax as a hint, to show that it's accessing a top-scope variable.

### Using the `$facts['fact_name']` hash syntax

Alternatively, facts are structured in a `$facts` hash, and your manifest code can access them as `$facts['fact_name']`. The variable name `$facts` is reserved, so local scopes cannot re-use it. Structured facts show up as a nested structure inside the `$facts` namespace, and can be accessed using Puppet's normal hash access syntax.

For example:

```
if $facts['os']['family'] == 'RedHat' {
  # ...
}
```

Accessing facts using this syntax makes for more readable and maintainable code, by making facts visibly distinct from other variables. It eliminates confusion that is possible when you use a local variable whose name happens to match that of a common fact.

Because of ambiguity with function invocation, the dot-separated access syntax that is available in Facter commands is not available with the `$facts` hash access syntax. However, you can instead use the `fact` function included in the `stdlib` module. Read more about it in the `stdlib` module README.

### Improving performance by blocking or caching built-in facts

If Facter is slowing down your code, you can configure Facter to block or cache built-in facts. When a system has a lot of something — for example, mount points or disks — Facter can take a long time to collect the facts from each one. When this is a problem, you can speed up Facter’s collection by configuring these settings in the `facter.conf` file:

- `blocklist` for blocking built-in facts you’re uninterested in.
- `ttls` for caching built-in facts you don’t need retrieved frequently.

### Related information

Scope on page 369
A scope is a specific area of code that is partially isolated from other areas of code.

Variables on page 204

Variables store values so that those values can be accessed in code later.

Hashes on page 311
Hashes map keys to values, maintaining the order of the entries according to insertion order.

Configuring Facter with facter.conf on page 457
The `facter.conf` file is a configuration file that allows you to cache and block fact groups, and manage how Facter interacts with your system. There are three sections: `facts`, `global`, and `cli`. All sections are optional and can be listed in any order within the file.

## Built-in variables

In addition to Facter's core facts and custom facts, Puppet creates several variables for a node to facilitate managing it. These variables are called trusted facts, server facts, agent facts, master variables, and compiler variables.

### Trusted facts

Normal facts are self-reported by the node, and nothing guarantees their accuracy. Trusted facts are extracted from the node's certificate, which can prove that the certificate authority checked and approved them, making them useful for deciding whether a given node should receive the sensitive data in its catalog.

Trusted facts is a hash that contains trusted data from the node's certificate. You can access the data using the syntax `$trusted['fact_name']`. The variable name `$trusted` is reserved, so local scopes cannot reuse it.

| Keys in the `$trusted` hash | Possible values |
|---|---|
| `authenticated` | An indication of whether the catalog request was authenticated, as well as how it was authenticated. The value will be one of these:<br>• `remote` for authenticated remote requests, as with agent-master Puppet configurations.<br>• `local` for all local requests, as with standalone Puppet apply nodes.<br>• `false` for unauthenticated remote requests, possible if your `auth.conf` configuration allows unauthenticated catalog requests. |
| `certname` | The node's subject certificate name, as listed in its certificate. When first requesting its certificate, the node requests a subject certificate name matching the value of its `certname` setting.<br>• If `authenticated` is `remote`, the value is the subject certificate name extracted from the node's certificate.<br>• If `authenticated` is `local`, the value is read directly from the `certname` setting.<br>• If `authenticated` is `false`, the value will be an empty string. |
| `domain` | The node's domain, as derived from its validated certificate name. The value can be empty if the certificate name doesn't contain a fully qualified domain name. |

| Keys in the `$trusted` hash | Possible values |
| --- | --- |
| `extensions` | A hash containing any custom extensions present in the node's certificate. The keys of the hash are the extension OIDs. OIDs in the ppRegCertExt range appear using their short names, and other OIDs appear as plain dotted numbers. If no extensions are present, or `authenticated` is `local` or `false`, this is an empty hash. |
| `hostname` | The node's hostname, as derived from its validated certificate name |

A typical `$trusted` hash looks something like this:

```
{
  'authenticated' => 'remote',
  'certname'      => 'web01.example.com',
  'domain'        => 'example.com',
  'extensions'    => {
                      'pp_uuid'                         => 'ED803750-
E3C7-44F5-BB08-41A04433FE2E',
                      'pp_image_name'           => 'storefront_production'
                      '1.3.6.1.4.1.34380.1.2.1' => 'ssl-termination'
                   },
  'hostname'      => 'web01'
}
```

Here is some example Puppet code using a certificate extension:

```
if $trusted['extensions']['pp_image_name'] == 'storefront_production' {
  include private::storefront::private_keys
}
```

Here's an example of a `hiera.yaml` file using certificate extensions in a hierarchy:

```
---
version: 5
hierarchy:
  - name: "Certname"
    path: "nodes/%{trusted.certname}.yaml"
  - name: "Original VM image name"
    path: "images/%{trusted.extensions.pp_image_name}.yaml"
  - name: "Machine role (custom certificate extension)"
    path: "role/%{trusted.extensions.'1.3.6.1.4.1.34380.1.2.1'}.yaml"
  - name: "Common data"
    path: "common.yaml"
```

### Server facts

The `$server_facts` variable provides a hash of server-side facts that cannot be overwritten by client side facts. This is important because it enables you to get trusted server facts that could otherwise be overwritten by client-side facts.

For example, the Puppet master sets the global `$::environment` variable to contain the name of the node's environment. However, if a node provides a fact with the name `environment`, that fact's value overrides the server-set `environment` fact. The same happens with other server-set global variables, like `$::servername` and `$::serverip`. As a result, modules can't reliably use these variables for whatever their intended purpose was.

A warning is issued any time a node parameter is overwritten.

Here is an example `$server_facts` hash:

```
{
  serverversion => "4.1.0",
  servername    => "v85ix8blah.delivery.example.com",
  serverip      => "192.0.2.10",
  environment   => "production",
}
```

### Agent facts

Puppet agent and Puppet apply both add several extra pieces of info to their facts before requesting or compiling a catalog. Like other facts, these are available as either top-scope variables or elements in the `$facts` hash.

| Agent facts | Values |
|---|---|
| `$clientcert` | The node's `certname` setting. This is self-reported; for the verified certificate name, use`$trusted['certname']`. |
| `$clientversion` | The current version of Puppet agent. |
| `$puppetversion` | The current version of Puppet on the node. |
| `$clientnoop` | The value of the node's `noop` setting (true or false) at the time of the run. |
| `$agent_specified_environment` | The value of the node's `environment` setting. If the master's node classifier specified an environment for the node, `$agent_specified_environment` and `$environment` can have different values. If no value was set for the `environment` setting (in `puppet.conf` or with `--environment`), the value of `$agent_specified_environment`is `undef`. That is, it doesn't default to `production` like the setting does. |

### Master variables

Several variables are set by the Puppet master. These are most useful when managing Puppet with Puppet, for example, managing the `puppet.conf` file with a template. Master variables are **not** available in the `$facts` hash.

| Master variables | Values |
|---|---|
| `$environment` (also available to`puppet apply`) | The agent node's environment. Note that nodes can accidentally or purposefully override this with a custom fact; the `$server_facts['environment']` variable always contains the correct environment, and can't be overridden. |
| `$servername` | The master's fully-qualified domain name (FQDN). Note that this information is gathered from the master by Facter, rather than read from the config files. Even if the master's certname is set to something other than its FQDN, this variable still contains the server's FQDN. |
| `$serverip` | The master's IP address. |
| `$serverversion` | The current version of Puppet on the master. |

| Master variables | Values |
| --- | --- |
| `$settings::<SETTING_NAME>`(also available to `puppet apply`) | The value of any of the master's configuration settings. This is implemented as a special namespace and these variables must be referred to by their qualified names. Other than `$environment` and `$clientnoop`, the agent node's settings are not available in manifests. If you wish to expose them to the master, you must create a custom fact. |
| `$settings::all_local` | Contains all variables in the `$settings`namespace as a hash of `<SETTING_NAME> => <SETTING_VALUE>`. This helps you reference settings that might be missing, because a direct reference to such a missing setting raises an error when `--strict_variables`is enabled. |

### Compiler variables

Compiler variables are set in every local scope by the compiler during compilation. They are mostly used when implementing complex defined types. Compiler variables are **not** available in the `$facts` hash.

These variables are always considered defined, suc `strict_variables` setting always considers these variables to be defined, but their value is `undef` whenever no other value is applicable.

| Compiler variables | Values |
| --- | --- |
| `$module_name` | The name of the module that contains the current class or defined type. |
| `$caller_module_name` | The name of the module in which the *specific instance* of the surrounding defined type was declared. This is useful when creating versatile defined types that will be reused by several modules. |

### Related information

CSR attributes and certificate extensions on page 491
When Puppet agent nodes request their certificates, the certificate signing request (CSR) usually contains only their certname and the necessary cryptographic information. Agents can also embed additional data in their CSR, useful for policy-based autosigning and for adding new trusted facts.

Configuring Hiera on page 403
The Hiera configuration file is called `hiera.yaml`. It configures the hierarchy for a given layer of data.

About environments on page 92
An environment is a branch that gets turned into a directory on your master.

Puppet settings on page 59
Customize Puppet settings in the main configuration file, called `puppet.conf`.

Custom facts on page 444
A typical fact in Facter is a collection of several elements, and is written either as a simple value ("flat" fact) or as structured data ("structured" fact).

Scope on page 369
A scope is a specific area of code that is partially isolated from other areas of code.

Defined resource types on page 227

*Defined resource types*, sometimes called defined types or defines, are blocks of Puppet code that can be evaluated multiple times with different parameters.

# Reserved words and acceptable names

You can use only certain characters for naming variables, modules, classes, defined types, and other custom constructs. Additionally, some words in the Puppet language are reserved and cannot be used as bare word strings or names.

## Reserved words

Reserved words cannot be used as:

- Bare word strings—to use these words as strings, you must enclose them in quotes.
- Names for custom functions.
- Names for classes.
- Names for custom resource types or defined resource types.

In addition, do not:

- Use the name of any existing resource type or function as the name of a function.
- Use the name of any existing resource type as the name of a defined type.
- Use the name of any existing data type (such as `integer`) as the name of a defined type.

**Table 1:**

| Reserved word | Description |
|---|---|
| `and` | Expression operator |
| `application` | Language keyword |
| `attr` | Reserved for future use |
| `case` | Language keyword |
| `component` | Reserved |
| `consumes` | Language keyword |
| `default` | Language keyword |
| `define` | Language keyword |
| `elsif` | Language keyword |
| `environment` | Reserved for symbolic namespace use |
| `false` | Boolean value |
| `function` | Language keyword |
| `if` | Language keyword |
| `import` | Former language keyword |
| `in` | Expression operator |
| `inherits` | Language keyword |
| `node` | Language keyword |

| Reserved word | Description |
|---|---|
| `or` | Expression operator |
| `private` | Reserved for future use |
| `produces` | Language keyword |
| `regexp` | Reserved |
| `site` | Language keyword |
| `true` | Boolean value |
| `type` | Language keyword |
| `undef` | Special value |
| `unit` | Reserved |
| `unless` | Language keyword |

### Reserved class names

Puppet automatically creates two names that must not be used as class names elsewhere:

- `main`: Puppet creates a `main` class, which contains any resources not contained by any other class.
- `settings`: Puppet creates a `settings` namespace, which contains variables with the settings available to the master.

Additionally, the names of data types can't be used as class names:

- `any`, `Any`
- `array`, `Array`
- `boolean`, `Boolean`
- `catalogentry`, `catalogEntry`, `CatalogEntry`
- `class`, `Class`
- `collection`, `Collection`
- `callable`, `Callable`
- `data`, `Data`
- `default`, `Default`
- `enum`, `Enum`
- `float`, `Float`
- `hash`, `Hash`
- `integer`, `Integer`
- `numeric`, `Numeric`
- `optional`, `Optional`
- `pattern`, `Pattern`
- `resource`, `Resource`
- `runtime`, `Runtime`
- `scalar`, `Scalar`
- `string`, `String`
- `struct`, `Struct`
- `tuple`, `Tuple`
- `type`, `Type`
- `undef`, `Undef`
- `variant`, `Variant`

### Reserved variable names

The following variable names are reserved. Unless otherwise noted, you can't assign values to them or use them as parameters in classes or defined types.

**Table 2:**

| Reserved variable name | Description |
|---|---|
| `$0`, `$1`, and every other variable name consisting only of digits | These are regex capture variables automatically set by regular expression used in conditional statements. Their values do not persist oustide their associated code block or selector value. Assigning these variables causes an error. |
| Top-scope Puppet built-in variables and facts | Built-in variables and facts are reserved at top scope, but you can safely reuse them at node or local scope. See built-in variables and facts for a list of these variables and facts. |
| `$facts` | Reserved for facts and cannot be reassigned at local scopes. |
| `$trusted` | Reserved for facts and cannot be reassigned at local scopes. |
| `$server_facts` | If enabled, this variable is reserved for trusted server facts and cannot be reassigned at local scopes. |
| `title` | Reserved for the title of a class or defined type. |
| `name` | Reserved for the name of a class or defined type. |

### Related information

Facts and built-in variables on page 254
Before requesting a catalog for a managed node, or compiling one with `puppet apply`, Puppet collects system information, called *facts*, by using the Facter tool. The facts are assigned as values to variables that you can use anywhere in your manifests. Puppet also sets some additional special variables, called *built-in variables*, which behave a lot like facts.

Scope on page 369
A scope is a specific area of code that is partially isolated from other areas of code.

## Acceptable characters in names

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

**CAUTION:** In some cases, names containing unsupported characters might still work. Such cases are bugs and could cease to work at any time. Removal of these bug cases is not limited to major releases.

### Classes and defined resource type names

The names of classes and defined resource types can consist of one or more *namespace* segments. Each namespace segment:

- Must begin with a lowercase letter.
- Can include lowercase letters.
- Can include digits.
- Can include underscores.

When you follow these rules, each namespace segment matches 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 are joined together in a class or defined 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
```

Some words and class names are reserved and cannot be used as class or defined type names. Additionally, the filename `init.pp` is reserved for the class named after any given module, so you cannot use the name `<MODULE NAME>::init` for a class or defined type.

### Variable names

Variable names are case-sensitive and must begin with a dollar sign (`$`). Most variable names must start with a lowercase letter or an underscore. The exception is regex capture variables, which are named with only numbers.

Variable names can include:

- Uppercase and lowercase letters
- Numbers
- Underscores ( _ ). If the first character is an underscore, access that variable only from its own local scope.

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`.

Optionally, the name of the very first namespace can be empty, representing the top namespace. The main reason to namespace this way is to indicate to anyone reading your code that you're accessing a top-scope variable, such as `$::is_virtual`.

You can also use a regular expression for variable names. Short variable names match the following regular expression:

```
\A\$[a-z0-9_][a-zA-Z0-9_]*\Z
```

Qualified variable names match the following regular expression:

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

### Module names

Module names obey the same rules as individual namespace segments, just as in a class or defined type name. That is, each namespace segment:

- Must begin with a lowercase letter.
- Can include lowercase letters.
- Can include digits.
- Can include underscores.

When you follow these rules, each namespace segment matches the following regular expression:

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

Reserved words and class names cannot be used as module names.

### Parameter names

Parameter names begin with a dollar sign prefix ($). The parameter name after the prefix:

- Must begin with a lowercase letter.
- Can include lowercase letters.
- Can include digits.
- Can include underscores.

When you follow these rules, a parameter name matches the following regular expression:

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

### Tag names

Tag names must begin with:

- A lowercase letter, or
- An number, or
- An underscore.

Tag names can include:

- Lowercase letters
- Uppercase letters
- Digits
- Underscores
- Colons
- Periods
- Hyphens

When you follow these rules, a tag name matches the following regular expression:

```
\A[[:alnum:]_][[:alnum:]_:.-]*\Z
```

### Resource names

Resource titles can contain any characters whatsoever and are case-sensitive. Resource names, or the *namevar* attribute, might be limited by the system being managed. For example, most operating systems have limits on the characters permitted in the name of a user account. You are generally responsible for knowing the name limits on the platforms you manage.

### Node names

Node names can contain:

- Letters
- Digits
- Periods
- Underscores
- Dashes

That is, node names match the regular expression:

```
 /\A[a-z0-9._-]+\Z/
```

### Environment names

Environment names can contain:

- Lowercase letters
- Numbers
- Underscores

That is, environment names match the regular expression:

```
\A[a-z0-9_]+\Z
```

## Custom resources

A *resource* is the basic unit that is managed by Puppet. Each resource has a set of attributes describing its state. Some attributes can be changed throughout the lifetime of the resource, whereas others are only reported back but cannot be changed, and some can only be set one time during initial creation.

A custom resource allows you to interact with something external. Three common use cases for this are:

- Parsing a file
- Running a command line tool
- Communicating with an API

To gather information about a resource and to enact changes on it, Puppet requires a *provider* to implement interactions. The provider can have parameters that influence its operation. To describe all these parts to the infrastructure and the consumers, the resource *type* defines all the metadata, including the list of the attributes. The provider contains the code to `get` and `set` the system state.

If you are starting from scratch, or want a simple method for writing types and providers, use the Resource API. Built on top of Puppet core, the Resource API makes the type and provider development easier, cheaper, safer, faster, and better. If you need to maintain existing code, need multiple providers, or need access to the catalog, use the old low-level types and providers method.

- Develop types and providers with the Resource API on page 265
The recommended method to create custom types and providers is to use the Resource API, which is built on top of Puppet core. It is easier, faster, and safer than the old types and providers method.
- Resource API reference on page 268
Use this information to understand how the Resource API works: how the resource is defined in the type, how resource management is implemented in the provider, and some of the known limitations of the Resource API.
- Low-level method for developing types and providers on page 283
You can define custom resource types for managing your infrastructure by defining types and providers. This original method has largely been replaced by the Resource API method, but you might already have a setup that uses this low-level method, or your situation might fall into one of the Resource API limitations.

### Develop types and providers with the Resource API

The recommended method to create custom types and providers is to use the Resource API, which is built on top of Puppet core. It is easier, faster, and safer than the old types and providers method.

To get started developing types and providers with the Resource API:

1. Download Puppet Development Kit (PDK) appropriate to your operating system and architecture.
2. Create a new module with PDK, or work with an existing PDK-enabled module. To create a new module, run `pdk new module <MODULE_NAME>` from the command line, specifying the name of the module. Respond to the dialog questions.
3. To add the `puppet-resource_api` gem and enable modern rspec-style mocking, open the `sync.yml` file in your editor, and add the following content:

```
# .sync.yml
---
Gemfile:
  optional:
```

```
    ':development':
      - gem: 'puppet-resource_api'
spec/spec_helper.rb:
  mock_with: ':rspec'
```

4. Apply these changes by running `pdk update`
5. To create the required files for a new type and provider in the module, run: `pdk new provider <provider_name>`

   You will get the following response:

   ```
   $ pdk new provider foo
   pdk (INFO): Creating '.../example/lib/puppet/type/foo.rb'from template.
   pdk (INFO): Creating '.../example/lib/puppet/provider/foo/foo.rb' from
    template.
   pdk (INFO): Creating '.../example/spec/unit/puppet/provider/foo/
   foo_spec.rb' from template.
   $
   ```

   The three generated files are the type (resource definition), the provider (resource implementation), and the unit tests. The default template contains an example that demonstrates the basic workings of the Resource API. This allows the unit tests to run immediately after creating the provider, which looks like this:

   ```
   $ pdk test unit
   [✓] Preparing to run the unit tests.
   [✓] Running unit tests.
   Evaluated 4 tests in 0.012065973 seconds: 0 failures, 0 pending.
   [✓] Cleaning up after running unit tests.
   $
   ```

#### Writing the type and provider

Write a *type* to describe the resource and define its metadata, and a *provider* to gather information about the resource and implement changes.

#### Writing the type

The type contains the shape of your resources. The template provides the necessary `name` and `ensure` attributes. You can modify their description and the name's type to match your resource. Add more attributes as you need.

```
# lib/puppet/type/yum.rb
require 'puppet/resource_api'

Puppet::ResourceApi.register_type(
  name: 'yum',
  docs: <<-EOS,
      This type provides Puppet with the capabilities to manage ...
    EOS
  attributes: {
    ensure: {
      type:    'Enum[present, absent]',
      desc:    'Whether this apt key should be present or absent on the
 target system.',
      default: 'present',
    },
    name: {
      type:      'String',
      desc:      'The name of the resource you want to manage.',
      behaviour: :namevar,
    },
  },
)
```

The following keys are available for defining attributes:

- `type`: the Puppet 4 data type allowed in this attribute. You can use all data types matching `Scalar` and `Data`.
- `desc`: a string describing this attribute. This is used in creating the automated API docs with puppet-strings.
- `default`: a default value used by the runtime environment; when the caller does not specify a value for this attribute.
- `behaviour` / `behavior`: how the attribute behaves. Available values include:
  - `namevar`: marks an attribute as part of the primary key or identity of the resource. A given set of `namevar` values must distinctively identify an instance.
  - `init_only`: this attribute can only be set during the creation of the resource. Its value is reported going forward, but trying to change it later leads to an error. For example, the base image for a VM or the UID of a user.
  - `read_only`: values for this attribute are returned by `get()`, but `set()` is not able to change them. Values for this should never be specified in a manifest. For example, the checksum of a file, or the MAC address of a network interface.
  - `parameter`: values for this attributes are not returned by `get()`. You can use this attribute to influence how the provider behaves. For example, you can influence the managehome attribute when creating a user.

#### Writing the provider

The provider is the most important part of your new resource, as it reads and enforces state. When you generate a provider with the `pdk new provider` command, PDK generates a provider file like this generated `yum.rb` file

```
require 'puppet/resource_api/simple_provider'

# Implementation for the yum type using the Resource API.
class Puppet::Provider::provider_name::Yum <
 Puppet::ResourceApi::SimpleProvider
  def get(_context)
    [
      {
        name: 'foo',
        ensure: 'present',
      },
      {
        name: 'bar',
        ensure: 'present',
      },
    ]
  end

  def create(context, name, should)
    context.notice("Creating '#{name}' with #{should.inspect}")
  end

  def update(context, name, should)
    context.notice("Updating '#{name}' with #{should.inspect}")
  end

  def delete(context, name)
    context.notice("Deleting '#{name}'")
  end
end
```

The optional `initialize` method can be used to set up state that is available throughout the execution of the catalog. This is most often used for establishing a connection when talking to a service, such as when you are managing a database.

The `get(context)` method returns a list of hashes describing the resources that are on the target system. The basic example would return an empty list. For example, these resources could be returned from this:

```
[
  {
    name: 'a',
    ensure: 'present',
  },
  {
    name: 'b',
    ensure: 'present',
  },
]
```

The `create`, `update`, and `delete` methods are called by the `SimpleProvider` base class to change the system as requested by the catalog. The `name` argument is the name of the resource that is being processed. `should` contains the attribute hash — in the same format as `get` returns — with the values in the catalog.

When adding Ruby code to a module, follow these guidelines:

- For small sized blocks of code, put the code in the provider.
- For medium sized blocks of code, put the code into a separate file in `lib/puppet_x/$forgeuser/$modulename.rb`. `puppet_x` is optional, but it helps keep the file name unique and reduces the risk of a file overwriting code from an unknown dependency.
- For large sized blocks of code, put the code in a separate gem.

### Unit testing

The generated unit tests in `spec/unit/puppet/provider/<PROVIDER_NAME>_spec.rb` are evaluated when you run `pdk test unit`.

## Resource API reference

Use this information to understand how the Resource API works: how the resource is defined in the type, how resource management is implemented in the provider, and some of the known limitations of the Resource API.

### Resource definition: the type

A type is a definition of a resource that Puppet can manage. The definition contains the resource's configurable properties and the parameters used to access it.

To make the resource known to the Puppet ecosystem, its definition, or *type* needs to be registered with Puppet. For example:

```
Puppet::ResourceApi.register_type(
  name: 'apt_key',
  desc: <<-EOS,
    This type provides Puppet with the capabilities to manage GPG keys
 needed
    by apt to perform package validation. Apt has it's own GPG keyring that
 can
    be manipulated through the `apt-key` command.

    apt_key { '6F6B15509CF8E59E6E469F327F438280EF8D349F':
      source => 'http://apt.puppetlabs.com/pubkey.gpg'
    }

    **Autorequires**:
    If Puppet is given the location of a key file which looks like an
 absolute
    path this type will autorequire that file.
  EOS
```

```
  attributes: {
    ensure: {
      type: 'Enum[present, absent]',
      desc: 'Whether this apt key should be present or absent on the target
 system.'
    },
    id: {
      type:      'Variant[Pattern[/\A(0x)?[0-9a-fA-F]{8}\Z/], Pattern[/
\A(0x)?[0-9a-fA-F]{16}\Z/], Pattern[/\A(0x)?[0-9a-fA-F]{40}\Z/]]',
      behaviour: :namevar,
      desc:      'The ID of the key you want to manage.',
    },
    source: {
      type: 'String',
      desc: 'Where to retrieve the key from, can be a HTTP(s) URL, or a
 local file. Files get automatically required.',
    },
    # ...
    created: {
      type:      'String',
      behaviour: :read_only,
      desc:      'Date the key was created, in ISO format.',
    },
  },
  autorequire: {
    file:    '$source', # evaluates to the value of the `source` attribute
    package: 'apt',
  },
)
```

The `Puppet::ResourceApi.register_type(options)` function takes the following keyword arguments:

- `name`: the name of the resource type.
- `desc`: a doc string that describes the overall working of the resource type, provides examples, and explains prerequisites and known issues.
- `attributes`: a hash mapping attribute names to their details. Each attribute is described by a hash containing the Puppet 4 data `type`, a `desc` string, a `default` value, and the `behavior` of the attribute: `namevar`, `read_only`, `init_only`, or a `parameter`.
  - `type`: the Puppet 4 data type allowed in this attribute.
  - `desc`: a string describing this attribute. This is used in creating the automated API docs with puppet-strings.
  - `default`: a default value that the runtime environment uses when you don't specify a value.
  - `behavior`/`behaviour`: how the attribute behaves. Currently available values:
    - `namevar`: marks an attribute as part of the "primary key" or "identity" of the resource. A given set of `namevar` values needs to distinctively identify an instance.
    - `init_only`: this attribute can only be set when creating the resource. Its value is reported going forward, but trying to change it later leads to an error. For example, the base image for a VM or the UID of a user.
    - `read_only`: values for this attribute are returned by `get()`, but `set()` is not able to change them. Values for this should never be specified in a manifest. For example, the checksum of a file, or the MAC address of a network interface.
    - `parameter`: these attributes influence how the provider behaves, and cannot be read from the target system. For example, the target file on inifile, or the credentials to access an API.
- `autorequire`, `autobefore`, `autosubscribe`, and `autonotify`: a hash mapping resource types to titles. The titles must either be constants, or, if the value starts with a dollar sign, a reference to the value of an attribute. If the specified resources exist in the catalog, Puppet creates the relationships that are requested here.
- `features`: a list of API feature names, specifying which optional parts of this spec the provider supports. Currently defined features: `canonicalize`, `simple_get_filter`, and `supports_noop`. See provider types for details.

### Composite namevars (`title_patterns`)

Each resource being managed must be identified by a unique title. Usually this is straightforward and a single attribute can be used to act as an identifier. But sometimes you need a composite of two attributes to uniquely identify the resource you want to manage.

If multiple attributes are defined with the namevar behavior, the type specifies `title_patterns` that tell the Resource API how to get at the attributes from the title. If `title_patterns` is not specified, a default pattern is applied and matches against the first declared namevar.

**Note:** The order of the `title_patterns` is important. You should declare the most specific pattern first and end with the most generic.

Each title pattern contains:

- `pattern`, which is a Ruby regular expression containing named captures. The names of the captures must be that of the namevar attributes.
- `desc`, a short description of what the pattern matches for.

For example:

```
Puppet::ResourceApi.register_type(
  name: 'software',
  docs: <<-DOC,
    This type provides Puppet with the capabilities to manage ...
  DOC
  title_patterns: [
    {
      pattern: %r{^(?<package>.*[^-])-(?<manager>.*)$},
      desc: 'Where the package and the manager are provided with a hyphen
 seperator',
    },
    {
      pattern: %r{^(?<package>.*)$},
      desc: 'Where only the package is provided',
    },
  ],
  attributes: {
    ensure: {
      type:    'Enum[present, absent]',
      desc:    'Whether this resource should be present or absent on the
 target system.',
      default: 'present',
    },
    package: {
      type:      'String',
      desc:      'The name of the package you want to manage.',
      behaviour: :namevar,
    },
    manager: {
      type:      'String',
      desc:      'The system used to install the package.',
      behaviour: :namevar,
    },
  },
)
```

These match the first title pattern:

```
software { php-yum:
  ensure=>'present'
}

software { php-gem:
```

```
  ensure=>'absent'
}
```

This matches the second title pattern:

```
software { php:
  manager='yum'
  ensure=>'present'
}
```

### Resource implementation: the provider

To make changes, a resource requires an implementation, or *provider*. It is the code used to retrieve, update and delete the resources of a certain type.

The two fundamental operations to manage resources are reading and writing system state. These operations are implemented as `get` and `set`. The implementation itself is a Ruby class in the `Puppet::Provider` namespace, named after the type using CamelCase.

**Note:** Due to the way Puppet autoload works, this is in a file called `puppet/provider/<type_name>/<type_name>.rb`. The class also has the CamelCased type name twice.

At runtime, the current and intended system states a specific resource. These are represented as Ruby hashes of the resource's attributes and applicable operational parameters:

```
class Puppet::Provider::AptKey::AptKey
  def get(context)
    [
      {
        name: 'name',
        ensure: 'present',
        created: '2017-01-01',
        # ...
      },
      # ...
    ]
  end

  def set(context, changes)
    changes.each do |name, change|
      is = change.has_key? :is ? change[:is] : get_single(name)
      should = change[:should]
      # ...
    end
  end
end
```

The `get` method reports the current state of the managed resources. It returns an enumerable of all existing resources. Each resource is a hash with attribute names as keys, and their respective values as values. It is an error to return values not matching the type specified in the resource type. If a requested resource is not listed in the result, it is considered to not exist on the system. If the `get` method raises an exception, the provider is marked as unavailable during the current run, and all resources of this type fails in the current transaction. The exception message is reported.

The `set` method updates resources to a new state. The `changes` parameter gets passed a hash of change requests, keyed by the resource's name. Each value is another hash with the optional `:is` and `:should` keys. At least one of the two must be specified. The values are of the same shape as those returned by `get`. After the `set`, all resources should be in the state defined by the `:should` values.

A missing `:should` entry indicates that a resource should be removed from the system. Even a type implementing the `ensure => [present, absent]` attribute pattern must react correctly on a missing `:should` entry.

An `:is` key might contain the last available system state from a prior `get` call. If the `:is` value is `nil`, the resources were not found by `get`. If there is no `:is` key, the runtime did not have a cached state available.

The `set` method should always return `nil`. Any progress signaling should be done through the logging utilities described below. If the `set` method throws an exception, all resources that should change in this call and haven't already been marked with a definite state, are marked as failed. The runtime only calls the `set` method if there are changes to be made, especially when resources are marked with `noop => true` (either locally or through a global flag). The runtime does not pass them to set. See `supports_noop` for changing this behavior if required.

Both methods take a context parameter which provides utilties from the runtime environment, and is described in more detail there.

### Implementing simple providers

In many cases, the resource type follows the conventional patterns of Puppet, and does not gain from the complexities around batch-processing changes. For those cases, the `SimpleProvider` class supplies a proven foundation that reduces the amount of code necessary to get going.

`SimpleProvider` requires that your type follows these common conventions:

- `name` is the name of your `namevar` attribute.
- `ensure` attribute is present and has the `Enum[absent, present]` type.

To start using `SimpleProvider`, inherit from the class like this:

```
require 'puppet/resource_api/simple_provider'

# Implementation for the wordarray type using the Resource API.
class Puppet::Provider::AptKey::AptKey < Puppet::ResourceApi::SimpleProvider
  # ...
```

Next, instead of the `set` method, the provider needs to implement the `create`, `update` or `delete` methods:

- `create(context, name, should)`: Called to create a resource.
  - `context`: provides utilities from the runtime environment.
  - `name`: the name of the new resource.
  - `should`: a hash of the attributes for the new instance.
- `update(context, name, should)`: Called to update a resource.
  - `context`: provides utilties from the runtime environment.
  - `name`: the name of the resource to change.
  - `should`: a hash of the desired state of the attributes.
- `delete(context, name)`: Called to delete a resource.
  - `context`: provides utilities from the runtime environment.
  - `name`: the name of the resource that should be deleted.

The `SimpleProvider` does basic logging and error handling.

#### Provider features

There are some use cases where an implementation provides a better experience than the default runtime environment provides. To avoid burdening the simplest providers with that additional complexity, these cases are hidden behind feature flags. To enable the special handling, the resource definition has a feature key to list all features implemented by the provider.

### `canonicalize`

Allows the provider to accept a wide range of formats for values without confusing the user.

```
Puppet::ResourceApi.register_type(
  name: 'apt_key',
```

```
  features: [ 'canonicalize' ],
)

class Puppet::Provider::AptKey::AptKey
  def canonicalize(context, resources)
    resources.each do |r|
      r[:name] = if r[:name].start_with?('0x')
                   r[:name][2..-1].upcase
                 else
                   r[:name].upcase
                 end
    end
  end
```

The runtime environment needs to compare user input from the manifest (the desired state) with values returned from `get` (the actual state), to determine whether or not changes need to be affected. In simple cases, a provider only accepts values from the manifest in the same format as `get` returns. No extra work is required, as a value comparison is enough. This places a high burden on the user to provide values in an unnaturally constrained format. In the example, the `apt_key` name is a hexadecimal number that can be written with, and without, the `'0x'` prefix, and the casing of the digits is irrelevant. A value comparison on the strings causes false positives if the user inputs format that does not match. There is no hexadecimal type in the Puppet language. To address this, the provider can specify the `canonicalize` feature and implement the `canonicalize` method.

The `canonicalize` method transforms its `resources` argument into the standard format required by the rest of the provider. The `resources` argument to `canonicalize` is an enumerable of resource hashes matching the structure returned by `get`. It returns all passed values in the same structure with the required transformations applied. It is free to reuse or recreate the data structures passed in as arguments. The runtime environment must use `canonicalize` before comparing user input values with values returned from `get`. The runtime environment always passes canonicalized values into `set`. If the runtime environment requires the original values for later processing, it protects itself from modifications to the objects passed into `canonicalize`, for example by creating a deep copy of the objects.

The context parameter is the same passed to `get` and `set`, which provides utilities from the runtime environment, and is described in more detail there.

**Note:** When the provider implements canonicalization, it aims to always log the canonicalized values. As a result of `get` and `set` producing and consuming canonically formatted values, it is not expected to present extra cost.

A side effect of these rules is that the canonicalization of the `get` method's return value must not change the processed values. Runtime environments may have strict or development modes that check this property.

For example, in the Puppet Discovery runtime environment it is bound to the `strict` setting, and follows the established practices:

```
# puppet resource --strict=error apt_key ensure=present
> runtime exception
```

```
# puppet resource --strict=warning apt_key ensure=present
> warning logged but values changed
```

```
# puppet resource --strict=off apt_key ensure=present
> values changed
```

### `simple_get_filter`

Allows for more efficient querying of the system state when only specific parts are required.

```
Puppet::ResourceApi.register_type(
  name: 'apt_key',
  features: [ 'simple_get_filter' ],
```

```
)

class Puppet::Provider::AptKey::AptKey
  def get(context, names = nil)
    [
      {
        name: 'name',
        # ...
      },
    ]
  end
```

Some resources are very expensive to enumerate. The provider can implement `simple_get_filter` to signal extended capabilities of the `get` method to address this. The provider's `get` method is called with an array of resource names, or `nil`. The `get` method must at least return the resources mentioned in the `names` array, but may return more. If the `names` parameter is `nil`, all existing resources should be returned. The `names` parameter defaults to `nil` to allow simple runtimes to ignore this feature.

The runtime environment calls `get` with a minimal set of names, and keeps track of additional instances returned to avoid double querying. To gain the most benefits from batching implementations, the runtime minimizes the number of calls into `get`.

#### supports_noop

When a resource is marked with `noop => true`, either locally or through a global flag, the standard runtime produces the default change report with a `noop` flag set. In some cases, an implementation provides additional information, for example commands that would get executed, or require additional evaluation before determining the effective changes, such as `exec`'s `onlyif` attribute. The resource type specifies the `supports_noop` feature to have `set` called for all resources, even those flagged with `noop`. When the `noop` parameter is set to `true`, the provider must not change the system state, but only report what it would change. The `noop` parameter should default to `false` to allow simple runtimes to ignore this feature.

```
Puppet::ResourceApi.register_type(
  name: 'apt_key',
  features: [ 'supports_noop' ],
)

class Puppet::Provider::AptKey::AptKey
  def set(context, changes, noop: false)
    changes.each do |name, change|
      is = change.has_key? :is ? change[:is] : get_single(name)
      should = change[:should]
      # ...
      do_something unless noop
    end
  end
end
```

#### remote_resource

Declaring this feature restricts the resource from being run locally. It is expected to execute all external interactions through the `context.transport` instance. The way that an instance is set up is runtime specific. Use `puppet/resource_api/transport/wrapper` as the base class for all devices:

```
# lib/puppet/type/nx9k_vlan.rb
Puppet::ResourceApi.register_type(
  name: 'nx9k_vlan',
  features: [ 'remote_resource' ],
  # ...
)
```

```
# lib/puppet/util/network_device/nexus/device.rb
require 'puppet'
require 'puppet/resource_api/transport/wrapper'
# force registering the transport schema
require 'puppet/transport/schema/device_type'

module Puppet::Util::NetworkDevice::Nexus
  class Device < Puppet::ResourceApi::Transport::Wrapper
    def initialize(url_or_config, _options = {})
      super('nexus', url_or_config)
    end
  end
end

# lib/puppet/provider/nx9k_vlan/nx9k_vlan.rb
class Puppet::Provider::Nx9k_vlan::Nx9k_vlan
  def set(context, changes, noop: false)
    changes.each do |name, change|
      is = change.has_key? :is ? change[:is] : get_single(name)
      should = change[:should]
      # ...
      context.transport.do_something unless noop
    end
  end
end
```

### Runtime environment

The primary runtime environment for the provider is the Puppet agent, a long-running daemon process. The provider can also be used in the `puppet apply` command, a self contained version of the agent, or the `puppet resource` command, a short-lived command line interface (CLI) process for listing or managing a single resource type. Other callers that want to access the provider must imitate these environments.

The primary life cycle of resource management in each of these tools is the transaction, a single set of changes, for example a catalog or a CLI invocation. The provider's class is instantiated one time for each transaction. Within that class the provider defines any number of helper methods to support itself. To allow for a transaction to set up the prerequisites for a provider and be used immediately, the provider is instantiated as late as possible. A transaction usually calls `get` one time, and may call `set` any number of times to make changes.

The object instance that hosts the `get` and `set` methods can be used to cache ephemeral state during execution. The provider should not try to cache state outside of its instances. In many cases, such caching won't help as the hosting process only manages a single transaction. In long-running runtime environments like the agent, the benefit of the caching needs to be balanced with the cost of the cache at rest, and the lifetime of cache entries, which are only useful when they are longer than the regular `runinterval`.

The runtime environment has the following utilities to provide a uniform experience for its users.

### Logging and reporting utilities

The provider needs to signal changes, successes, and failures to the runtime environment. The `context` is the primary way to do this. It provides a structured logging interface for all provider actions. Using this information, the runtime environments can do automatic processing, emit human readable progress information, and provide status messages for operators.

To provide feedback about the overall operation of the provider, the `context` has the usual set of loglevel methods that take a string, and pass that up to the runtime environments logging infrastructure. For example:

```
context.warning("Unexpected state detected, continuing in degraded mode.")
```

Results in the following message:

```
Warning: apt_key: Unexpected state detected, continuing in degraded mode.
```

Other common messages include:

- `debug`: Detailed messages to understand everything that is happening at runtime, shown on request.
- `info`: Regular progress and status messages, especially useful before long-running operations, or before operations that can fail, to provide context for interactive users.
- `notice`: Indicates state changes and other events of notice from the regular operations of the provider.
- `warning`: Signals error conditions that do not (yet) prohibit execution of the main part of the provider; for example, deprecation warnings, temporary errors.
- `err`: Signals error conditions that have caused normal operations to fail.
- `critical`, `alert`, `emerg`: should not be used by resource providers.

In simple cases, a provider passes off work to an external tool, logs the details there, and then reports back to Puppet acknowledging these changes. This is called resource status signaling, and looks like this:

```
@apt_key_cmd.run(context, action, key_id)
context.processed(key_id, is, should)
```

It reports all changes from `is` to `should`, using default messages.

Providers that want to have more control over the logging throughout the processing can use the more specific `created(title)`, `updated(title)`, `deleted(title)`, `unchanged(title)` methods. To report the change of an attribute, the `context` provides a `attribute_changed(title, attribute, old_value, new_value, message)` method.

Most of those messages are expected to be relative to a specific resource instance, and a specific operation on that instance. To enable detailed logging without repeating key arguments, and to provide consistent error logging, the `context` provides logging context methods to capture the current action and resource instance:

```
context.updating(title) do
  if apt_key_not_found(title)
    context.warning('Original key not found')
  end

  # Update the key by calling CLI tool
  apt_key(...)

  context.attribute_changed('content', nil, content_hash,
    message: "Replaced with content hash #{content_hash}")
end
```

This results in the following messages:

```
Debug: Apt_key[F1D2D2F9]: Started updating
Warning: Apt_key[F1D2D2F9]: Updating: Original key not found
Debug: Apt_key[F1D2D2F9]: Executing 'apt-key ...'
Debug: Apt_key[F1D2D2F9]: Successfully executed 'apt-key ...'
Notice: Apt_key[F1D2D2F9]: Updating content: Replaced with content hash
 E242ED3B
Notice: Apt_key[F1D2D2F9]: Successfully updated
```

In the case of an exception escaping the block, the error is logged appropriately:

```
Debug: Apt_keyF1D2D2F9]: Started updating
Warning: Apt_key[F1D2D2F9]: Updating: Original key not found
Error: Apt_key[F1D2D2F9]: Updating failed: Something went wrong
```

Logging contexts process all exceptions. A `StandardError` is assumed to be regular failures in handling resources, and are consumed after logging. Everything else is assumed to be a fatal application-level issue, and is passed up the stack, ending execution. See the Ruby documentation for details on which exceptions are not a `StandardError`.

The equivalent long-hand form of manual error handling:

```
context.updating(title)
begin
  unless title_got_passed_to_set(title)
    raise Puppet::DevError, 'Managing resource outside of requested set:
 %{title}')
  end

  if apt_key_not_found(title)
    context.warning('Original key not found')
  end

  # Update the key by calling CLI tool
  result = @apt_key_cmd.run(...)

  if result.exitstatus != 0
    context.error(title, "Failed executing apt-key #{...}")
  else
    context.attribute_changed(title, 'content', nil, content_hash,
      message: "Replaced with content hash #{content_hash}")
  end
  context.changed(title)
rescue Exception => e
  context.error(title, e, message: 'Updating failed')
  raise unless e.is_a? StandardError
end
```

This example is only for demonstration purposes. In the normal course of operations, providers should always use the utility functions.

The following methods are available:

- Block functions: these functions provide logging and timing around a provider's core actions. If the the passed `&block` returns, the action is recorded as successful. To signal a failure, the block should raise an exception explaining the problem:
  - `creating(titles, message: 'Creating', &block)`
  - `updating(titles, message: 'Updating', &block)`
  - `deleting(titles, message: 'Deleting', &block)`
  - `processing(title, is, should, message: 'Processing', &block)`: generic processing of a resource, produces default change messages for the difference between `is:` and `should:`.
  - `failing(titles, message: 'Failing', &block)`: unlikely to be used often, but provided for completeness. It always records a failure.
- Action functions:
  - `created(titles, message: 'Created')`
  - `updated(titles, message: 'Updated')`
  - `deleted(titles, message: 'Deleted')`
  - `processed(title, is, should)`: the resource has been processed. It produces default logging for the resource and each attribute
  - `failed(titles, message:)`: the resource has not been updated successfully

- Attribute Change notifications:
  - `attribute_changed(title, attribute, is, should, message: nil)`: notify the runtime environment that a specific attribute for a specific resource has changed. `is` and `should` are the original and the new value of the attribute. Either can be `nil`.
- Plain messages:
  - `debug(message)`
  - `debug(titles, message:)`
  - `info(message)`
  - `info(titles, message:)`
  - `notice(message)`
  - `notice(titles, message:)`
  - `warning(message)`
  - `warning(titles, message:)`
  - `err(message)`
  - `err(titles, message:)`

`titles` can be a single identifier for a resource or an array of values, if the following block batch processes multiple resources in one pass. If that processing is not atomic, providers should instead use the non-block forms of logging, and provide accurate status reporting on the individual parts of update operations.

A single `set()` execution may only log messages for instances that have been passed, as part of the `changes` to process. Logging for instances not requested to be changed causes an exception - the runtime environment is not prepared for other resources to change.

The provider is free to call different logging methods for different resources in any order it needs to. The only ordering restriction is for all calls specifying the same `title`. For example, the `attribute_changed` needs logged before that resource's action logging, and the `context` needs to be opened before any other logging for this resource.

### Type definition

The provider can gain insight into the type definition through these `context.type` utility methods:

- `attributes`: returns a hash containing the type attributes and it's properties.
- `ensurable?`: returns `true` if the type contains the ensure attribute.
- `feature?(feature)`: returns `true` if the type supports a given provider feature.

For example:

```
# example from simple_provider.rb

def set(context, changes)
  changes.each do |name, change|
  is = if context.type.feature?('simple_get_filter')
         change.key?(:is) ? change[:is] : (get(context, [name]) || []).find
 { |r| r[:name] == name }
       else
         change.key?(:is) ? change[:is] : (get(context) || []).find { |r|
 r[:name] == name }
       end
  ...

end
```

### Resource API transports

A *transport* connects providers to remote resources, such as a device, cloud infrastructure, or a REST API.

The transport class contains the code for managing connections and processing information to and from the remote resource. The transport schema, similar to a resource type, describes the structure of the data that is passed for it to make a connection.

### Transport implementation methods

When you are writing a transport class to manage remote resources, use the following methods as appropriate:

- `initialize(context, connection_info)`
  - The `connection_info` contains a hash that matches the schema. After you run the initialize method, the provider assumes that you have defined your transport in such as way as to ensure that it's ready for processing requests. The transport will report connection errors by throwing an exception, for example, if the network is unreachable or the credentials are rejected. In some cases, for example when the target is a REST API, no processing will happen during initialization.
- `verify(context)`
  - Use this method to check whether the transport can connect to the remote target. If the connection fails, the transport will raise an exception.
- `facts(context)`
  - Use this method to access the target and the facts hash which contains a subset of default facts from Facter, and more specific facts appropriate for the target.
- `close(context)`
  - Use this method to close the connection. Calling this method releases the transport so that you can't use it any more and frees up cache and operating system resources, such as open connections. For implementation quality, the library will ignore exceptions that are thrown.

**Note:** The `context` is the primary way to signal changes, successes, and failures to the runtime environment. For more information, see Runtime environment.

An example of a transport class:

```
# lib/puppet/transport/device_type.rb
module Puppet::Transport
  # The main connection class to a Device endpoint
  class DeviceType
    def initialize(context, connection_info)
      # Add additional validation for connection_info
      # and pre-load a connection if it is useful for your target
    end

    def verify(context)
      # Test that transport can talk to the remote target
    end

    def facts(context)
      # Access target, return a Facter facts hash
    end

    def close(context)
      # Close connection, free up resources
    end
  end
end
```

An example of a corresponding schema:

```
# lib/puppet/transport/device_type.rb
```

```
Puppet::ResourceAPI.register_transport(
  name: 'device_type', # points at class Puppet::Transport::DeviceType
  desc: 'Connects to a device_type',
  connection_info: {
    host: {
      type: 'String',
      desc: 'The host to connect to.',
    },
    user: {
      type: 'String',
      desc: 'The user.',
    },
    password: {
      type: 'String',
      sensitive: true,
      desc: 'The password to connect.',
    },
    enable_password: {
      type: 'String',
      sensitive: true,
      desc: 'The password escalate to enable access.',
    },
    port: {
      type: 'Integer',
      desc: 'The port to connect to.',
    },
  },
)
```

If the following attributes apply to your target, use these names for consistency across transports:

- `uri`: use to specify which URL to connect to.
- `host`: use to specify an IP or address to connect to.
- `protocol`: use to specify which protocol the transport should use, for example `http`, `https`, `ssh` or `tcp`.
- `user`: the user you want the transport to connect as.
- `port`: the port you want the transport to connect to.

Do not use the following keywords in when writing the `connection_info`:

- `name`
- `path`
- `query`
- `run-on`
- `remote-transport`
- `remote-*`
- `implementations`

To ensure that the data the schema passes to the implementation is handled securely, set password attributes to `sensitive: true`. Attributes marked with the `sensitive` flag allow a user interface based on this schema to make appropriate presentation choices, such as obscuring the password field. Values that you've marked sensitive are passed to the transport wrapped in the type `Puppet::Pops::Types::PSensitiveType::Sensitive`. This keeps the value from being logged or saved inadvertently while it is being transmitted between components. To access the sensitive value within the transport, use the `unwrap` method, for example, `connection_info[:password].unwrap`.

### Errors and retry handling in transport implementation

The Resource API does not put many constraints on when and how a transport can fail. The remote resource you are connecting to will have it's own device specific connection and error handling capabilities. Be aware of the following issues that may arise:

- Your initial connection might fail. To retry making the connection, verify whether you have network problems or whether there has been a service restart of the target that you are trying to connect to. As part of the retry logic, the transport avoids passing these issues to other parts of your system and waits up to 30 seconds for a single target to recover. When you execute a retry, the transport logs transient problems at the `notice` level.
- After you make your connection and have run the initialize method, the transport might apply deeper validation to the passed connection information — like mutual exclusivity of optional values, for example, `password` or `key` — and throw an `ArgumentError`. The transport then tries to establish a connection to the remote target. If this fails due to unrecoverable errors, it throws another exception.
- The `verify` and `facts` methods, like `initialize`, throw exceptions only when unrecoverable errors are encountered, or when the retry logic times out.

### Port your existing device code to transports

If you have old code that uses Device, you should port it by updating your code with the following replacements as appropriate:

- Move the device class to `Puppet::Transport`
- Change `Util::NetworkDevice::NAME::Device` to `ResourceApi::Transport::NAME`
- Change the initialization to accept and process a `connection_info` hash.
- When accessing the `connection_info` in your new transport, change all string keys to symbols, for example, `name` to `:name`.
- Add `context` as the first argument to your `initialize` and `facts` method.
- Change `puppet/util/network_device/NAME/device` to `puppet/transport/NAME`
- Replace calls to Puppet logging with calls to the context logger

**Note:** If you can't port your code at this time, your providers can still access your Device through `context.device`, but you won't be able to make use of the functionality in Bolt plans.

After porting your code, you will have a transport class and a shim `Device` class that connects your transport in a way that Puppet understands. Specify the transport name in the `super` call to make the connection:

```
# lib/puppet/type/nx9k_vlan.rb
Puppet::ResourceApi.register_type(
  name: 'nx9k_vlan',
  features: [ 'remote_resource' ],
  # ...
)

# lib/puppet/util/network_device/nexus/device.rb
require 'puppet/resource_api/transport/wrapper'
# force registering the transport
require 'puppet/transport/schema/nexus'

module Puppet::Util::NetworkDevice::Nexus
  class Device < Puppet::ResourceApi::Transport::Wrapper
    def initialize(url_or_config, _options = {})
      super('nexus', url_or_config)
    end
  end
end
```

**Note:** Agent versions 6.0 to 6.3 are incompatible with this way of executing remote content. These versions will not be supported after support for PE 2019.0 ends.

### Resource API limitations

This Resource API is not a full replacement for the original low-level types and providers method. Here is a list of the current limitations. If they apply to your situation, the low-level types and providers method might be a better solution. The goal of the new Resource API is not to be a replacement of the prior one, but to be a simplified way to get the same results for the majority of use cases.

### You can't have multiple providers for the same type

The low-level type and provider method allows multiple providers for the same resource type. This allows the creation of abstract resource types, such as packages, which can span multiple operating systems. Automatic selection of an OS-appropriate provider means less work for the user, as they don't have to address whether the package needs to be managed using apt or yum in their code .

Allowing multiple providers means more complexity and more work for the type or provider developer, including:

- attribute sprawl
- disparate feature sets between the different providers for the same abstract type
- complexity in implementation of both the type and provider pieces stemming from the two issues above

The Resource API does not implement support for multiple providers.

Should support for multiple providers be desirable for a given type, your options are:

- Use the older, more complex type and provider method, or
- Implement multiple similar types using the Resource API, and select the platform-appropriate type in Puppet code. For example:

```
define package (
  Ensure $ensure,
  Enum[apt, rpm] $provider, # have a hiera 5 dynamic binding to a function
 choosing a sensible default for the current system
  Optional[String] $source  = undef,
  Optional[String] $version = undef,
  Optional[Hash] $options   = { },
) {
  case $provider {
    apt: {
      package_apt { $title:
        ensure          => $ensure,
        source          => $source,
        version         => $version,
        *               => $options,
      }
    }
    rpm: {
      package_rpm { $title:
        ensure => $ensure,
        source => $source,
        *      => $options,
      }
      if defined($version) { fail("RPM doesn't support \$version") }
      # ...
    }
  }
}
```

### Only built-in Puppet 4 data types are available

Currently, only built-in Puppet 4 data types are usable. This is because the type information is required on the agent, but Puppet has not made it available yet. Even after that is implemented, modules have to wait until the functionality is widely available before being able to rely on it.

### There is no catalog access

There is no way to access the catalog from the provider. Several existing types rely on this to implement advanced functionality. Some of these use cases would be better off being implemented as "external" catalog transformations, instead of munging the catalog from within the compilation process.

#### No logging for unmanaged instances

Previously, the provider could provide log messages for resource instances that were not passed into the `set` call. In the current implementation, these causes an error.

#### Automatic relationships constrained to consts and attribute values

The Puppet 3 type API allows arbitrary code execution for calculating automatic relationship targets. The Resource API is more restrained, but allows understanding the type's needs by inspecting the metadata.

## Low-level method for developing types and providers

You can define custom resource types for managing your infrastructure by defining types and providers. This original method has largely been replaced by the Resource API method, but you might already have a setup that uses this low-level method, or your situation might fall into one of the Resource API limitations.

Using this types and providers method, you can add new resource types to Puppet. This section describes what types and providers are, how they interact, and how to develop them.

Low-level Puppet types and providers development is done in Ruby. Previous experience with Ruby is helpful. Alternatively, opt for the more straightforward Resource API method for developing types and providers.

The internals of how types are created have changed over Puppet's lifetime, and this documentation describes effective development methods, skipping over all the things you can but probably shouldn't do.

When making a new resource type, you create two things:

- The type definition, which is a model of the resource type. It defines what parameters are available, handles input validation, and determines what features a provider can (or should) provide.
- One or more providers for that type. The provider implements the type by translating its capabilities into specific operations on a system. For example, the `package` type has `yum` and `apt` providers which implement package resources on Red Hat-like and Debian-like systems, respectively.

### Deploying types and providers

To use new types and providers:

1. The type and providers must be present in a module on the master. Like other types of plug-in (such as custom functions and custom facts), they go in the module's `lib` directory:
   - Type files: `lib/puppet/type/<TYPE NAME>.rb`
   - Provider files: `lib/puppet/provider/<TYPE NAME>/<PROVIDER NAME>.rb`
2. If you are using an agent-master deployment, each agent node must have its `pluginsync` setting in `puppet.conf` set to `true`, which is the default. In masterless Puppet, using `puppet apply`, the `pluginsync` setting is not required, but the module that contains the type and providers must be present on each node.

- Type development on page 284

When you define a resource type, focus on what the resource can do, not how it does it.

- Provider development on page 290

Providers are back-ends that support specific implementations of a given resource type, particularly for different platforms. Not all resource types have or need providers, but any resource type concerned about portability will likely need them.

#### Related information

Plug-ins in modules on page 104

Puppet supports several kinds of plug-ins, which are distributed in modules. These plug-ins enable features such as custom facts and functions for managing your nodes. Modules that you download from the Forge can include these kinds of plug-ins, and you can also develop your own.

Custom resources on page 265

A *resource* is the basic unit that is managed by Puppet. Each resource has a set of attributes describing its state. Some attributes can be changed throughout the lifetime of the resource, whereas others are only reported back but cannot be changed, and some can only be set one time during initial creation.

Resource API limitations on page 281
This Resource API is not a full replacement for the original low-level types and providers method. Here is a list of the current limitations. If they apply to your situation, the low-level types and providers method might be a better solution. The goal of the new Resource API is not to be a replacement of the prior one, but to be a simplified way to get the same results for the majority of use cases.

### Type development

When you define a resource type, focus on what the resource can do, not how it does it.

**Note:** Unless you are maintaining existing type and provider code, or the Resource API limitations affect you, use the Resource API to create custom resource types, instead of this method.

#### Creating types

Types are created by calling the `newtype` method on the `Puppet::Type` class:

```
# lib/puppet/type/database.rb
Puppet::Type.newtype(:database) do
  @doc = "Create a new database."
  # ... the code ...
end
```

The name of the type is the only required argument to `newtype`. The name must be a Ruby symbol, and the name of the file containing the type must match the type's name.

The `newtype` method also requires a block of code, specified with either curly braces (`{ ... }`) or the `do ... end` syntax. The code block implements the type, and contains all of the properties and parameters. The block will not be passed any arguments.

You can optionally specify a self-refresh option for the type by putting `:self_refresh => true` after the name. Doing so causes resources of this type to refresh (as if they had received an event through a notify-subscribe relationship) whenever a change is made to the resource. A notable use of this option is in the core `mount` type.

#### Documenting types

Write a description for the custom resource type in the type's `@doc` instance variable. The description can be extracted by the `puppet doc --reference type` command, which generates a complete type reference which includes your new type, and by the `puppet describe` command, which outputs information about specific types.

Write the description as a string in standard Markdown format. When the Puppet tools extract the string, they strip the greatest common amount of leading whitespace from the front of each line, excluding the first line. For example:

```
Puppet::Type.newtype(:database) do
  @doc = %q{Creates a new database. Depending
    on the provider, this might create relational
    databases or NoSQL document stores.

    Example:

        database {'mydatabase':
          ensure => present,
          owner  => root,
        }
  }
end
```

In this example, any whitespace would be trimmed from the first line (in this case, it’s zero spaces), then the greatest common amount would be trimmed from remaining lines. Three lines have four leading spaces, two lines have six, and two lines have eight, so four leading spaces would be trimmed from each line. This leaves the example code block indented by four spaces, and thus doesn't break the Markdown formatting.

### Properties and parameters

The bulk of a type definition consists of properties and parameters, which become the resource attributes available when declaring a resource of the new type.

The difference between a property and a parameter is subtle but important:

- Properties correspond to something measurable on the target system. For example, the UID and GID of a user account are properties, because their current state can be queried or changed. In practical terms, setting a value for a property causes a method to be called on the provider.
- Parameters change how Puppet manages a resource, but do not necessarily map directly to something measurable. For example, the `user` type's `managehome` attribute is a parameter — its value affects what Puppet does, but the question of whether Puppet is managing a home directory isn't an innate property of the user account.

Additionally, there are a few special attributes called metaparameters, which are supported by all resource types. These don't need to be handled when creating new types; they're implemented elsewhere.

A type definition typically has multiple properties, and must have at least one parameter.

### Properties

A custom type's properties are at the heart of defining how the resource works. In most cases, it's the properties that interact with your resource's providers.

If you define a property named `owner`, then when you are retrieving the state of your resource, then the `owner` property calls the `owner` method on the provider. In turn, when you are setting the state (because the resource is out of sync), then the `owner` property calls the `owner=` method to set the state on disk.

There's one common exception to this: The `ensure` property is special because it's used to create and destroy resources. You can set this property up on your resource type just by calling the `ensurable` method in your type definition:

```
Puppet::Type.newtype(:database) do
  ensurable
  ...
end
```

This property uses three methods on the provider: `create`, `destroy`, and `exists?`. The last method, somewhat obviously, is a Boolean to determine if the resource exists. If a resource's `ensure` property is out of sync, then no other properties are checked or modified.

You can modify how `ensure` behaves, such as by adding other valid values and determining what methods get called as a result; see types like `package` for examples.

The rest of the properties are defined a lot like you define the types, with the `newproperty` method, which should be called on the type:

```
Puppet::Type.newtype(:database) do
  ensurable
  newproperty(:owner) do
    desc "The owner of the database."
    ...
  end
end
```

Note the call to `desc`; this sets the documentation string for this property, and for Puppet types that get distributed with Puppet, it is extracted as part of the Type reference.

When Puppet was first developed, there would typically be a lot of code in this property definition. Now, however, you only define valid values or set up validation and munging. If you specify valid values, then Puppet only accepts those values, and automatically handles accepting either strings or symbols. In most cases, you only define allowed values for `ensure`, but it works for other properties, too:

```
newproperty(:enable) do
  newvalue(:true)
  newvalue(:false)
end
```

You can attach code to the value definitions (this code would be called instead of the `property=` method), but it's normally unnecessary.

For most properties, though, it is sufficient to set up validation:

```
newproperty(:owner) do
  validate do |value|
    unless value =~ /^\w+/
      raise ArgumentError, "%s is not a valid user name" % value
    end
  end
end
```

Note that the order in which you define your properties can be important: Puppet keeps track of the definition order, and it always checks and fixes properties in the order they are defined.

### Customizing behavior

By default, if a property is assigned multiple values in an array:

- It is considered in sync if any of those values matches the current value.
- If none of those values match, the first one is used when syncing the property.

If, instead, the property should only be in sync if all values match the current value (for example, a list of times in a cron job), declare this:

```
newproperty(:minute, :array_matching => :all) do # :array_matching defaults
 to :first
  ...
end
```

You can also customize how information about your property gets logged. You can create an `is_to_s` method to change how the current values are described, `should_to_s` to change how the desired values are logged, and `change_to_s` to change the overall log message for changes. See current types for examples.

### Handling property values

When a resource is created with a list of desired values, those values are stored in each property in its `@should` instance variable. You can retrieve those values directly by calling `should` on your resource (although note that when `:array_matching` is set to `:first` you get the first value in the array; otherwise you get the whole array):

```
myval = should(:color)
```

When you're not sure (or don't care) whether you're dealing with a property or parameter, it's best to use `value`:

```
myvalue = value(:color)
```

### Parameters

Parameters are defined the same way as properties. The difference between them is that parameters never result in methods being called on providers.

To define a new parameter, call the `newparam` method. This method takes the name of the parameter (as a symbol) as its argument, as well as a block of code. You can and should provide documentation for each parameter by calling the `desc` method inside its block. Tools that generate docs from this description trim leading whitespace from multiline strings, as described for type descriptions.

```
newparam(:name) do
  desc "The name of the database."
end
```

### Namevar

Every type must have at least one mandatory parameter: the namevar. This parameter uniquely identifies each resource of the type on the target system — for example, the path of a file on disk, the name of a user account, or the name of a package.

If the user doesn't specify a value for the namevar when declaring a resource, its value defaults to the title of the resource.

There are three ways to designate a namevar. Every type must have exactly one parameter that meets exactly one of these criteria:

1. Create a parameter whose name is `:name`. Because most types just use `:name` as the namevar, it gets special treatment and automatically becomes the namevar.

   ```
   newparam(:name) do
     desc "The name of the database."
   end
   ```

2. Provide the `:namevar => true` option as an additional argument to the `newparam` call. This allows you to use a namevar with a different, more descriptive name, such as the `file` type's `path` parameter.

   ```
   newparam(:path, :namevar => true) do
     ...
   end
   ```

3. Call the `isnamevar` method (which takes no arguments) inside the parameter's code block. This allows you to use a namevar with a different, more descriptive name. There is no practical difference between this and option 2.

   ```
   newparam(:path) do
     isnamevar
     ...
   end
   ```

### Specifying allowed values

If your parameter has a fixed list of valid values, you can declare them all at the same time:

```
newparam(:color) do
  newvalues(:red, :green, :blue, :purple)
end
```

You can specify regular expressions in addition to literal values; matches against regex always happen after equality comparisons against literal values, and those matches are not converted to symbols. For instance, given the following definition:

```
newparam(:color) do
```

```
  desc "Your color, and stuff."

  newvalues(:blue, :red, /.+/)
end
```

If you provide blue as the value, then your parameter is set to `:blue`, but if you provide green, then it is set to `"green"`.

### Validation and munging

If your parameter does not have a defined list of values, or you need to convert the values in some way, you can use the validate and munge hooks:

```
newparam(:color) do
  desc "Your color, and stuff."

  newvalues(:blue, :red, /.+/)

  validate do |value|
    if value == "green"
      raise ArgumentError,
        "Everyone knows green databases don't have enough RAM"
    else
      super
    end
  end

  munge do |value|
    case value
    when :mauve, :violet # are these colors really any different?
      :purple
    else
      super
    end
  end
end
```

The default `validate` method looks for values defined using `newvalues` and if there are any values defined it accepts only those values (this is how allowed values are validated). The default `munge` method converts any values that are specifically allowed into symbols. If you override either of these methods, note that you lose this value handling and symbol conversion, which you'll have to call `super` for.

Values are always validated before they're munged.

Lastly, validation and munging only happen when a value is assigned. They have no role to play at all during use of a given value, only during assignment.

### Boolean parameters

Boolean parameters are common. To avoid repetition, some utilities are available:

```
require 'puppet/parameter/boolean'
# ...
newparam(:force, :boolean => true, :parent => Puppet::Parameter::Boolean)
```

There are two parts here. The `:parent => Puppet::Parameter::Boolean` part configures the parameter to accept lots of names for true and false, to make things easy for your users. The `:boolean => true` creates a `boolean` method on the type class to return the value of the parameter. In this example, the method would be named `force?`.

### Automatic relationships

Your type can specify automatic relationships it can have with resources.

You can use `autorequire`, `autobefore`, `autonotify`, and `autosubscribe`, which all require a resource type as an argument, and your code should return a list of resource names that your resource could be related to.

```
autorequire(:user) do
  self[:user]
end
```

This won't throw an error if resources with those names do not exist. The purpose of this hook is to make sure that if any required resources are being managed, they get applied before the requiring resource.

### Agent-side pre-run resource validation

A resource can have prerequisites on the target, without which it cannot be synced. In some cases, if the absence of these prerequisites would be catastrophic, you might want to halt the catalog run if you detect a missing prerequisite.

In this situation, define a method in your type named `pre_run_check`. This method can do any check you want. It should take no arguments, and should raise a `Puppet::Error` if the catalog run should be halted.

If a type has a `pre_run_check` method, Puppet agent and `puppet apply` runs the check for every resource of the type before attempting to apply the catalog. It collects any errors raised, and presents all of them before halting the catalog run.

As a trivial example, here's a pre-run check that fails randomly, about one time out of six:

```
Puppet::Type.newtype(:thing) do
  newparam :name, :namevar => true

  def pre_run_check
    if(rand(6) == 0)
      raise Puppet::Error, "Puppet roulette failed, no catalog for you!"
    end
  end
end
```

### How types and providers interact

The type definition declares the features that a provider must have and what's required to make them work. Providers can either be tested for whether they suffice, or they can declare that they have the features. Because a type's properties call getter and setter methods on the providers, the providers must define getters and setters for each property (except `ensure`).

Additionally, individual properties and parameters in the type can declare that they require one or more specific features, and Puppet throws an error if those parameters are used with providers missing those features:

```
newtype(:coloring) do
  feature :paint, "The ability to paint.", :methods => [:paint]
  feature :draw, "The ability to draw."

  newparam(:color, :required_features => %w{paint}) do
    ...
  end
end
```

The first argument to the feature method is the name of the feature, the second argument is its description, and after that is a hash of options that help Puppet determine whether the feature is available. The only option currently supported is specifying one or more methods that must be defined on the provider. If no methods are specified, then the provider needs to specifically declare that it has that feature:

```
Puppet::Type.type(:coloring).provide(:drawer) do
  has_feature :draw
```

```
end
```

The provider can specify multiple available features at the same time with `has_features`.

When you define features on your type, Puppet automatically defines the following class methods on the provider:

- `feature?`: Passed a feature name, returns true if the feature is available or false otherwise.
- `features`: Returns a list of all supported features on the provider.
- `satisfies?`: Passed a list of feature, returns true if they are all available, false otherwise.

Additionally, each feature gets a separate Boolean method, so the above example would result in a `paint?` method on the provider.

#### Related information

Provider development on page 290
Providers are back-ends that support specific implementations of a given resource type, particularly for different platforms. Not all resource types have or need providers, but any resource type concerned about portability will likely need them.

### Provider development

Providers are back-ends that support specific implementations of a given resource type, particularly for different platforms. Not all resource types have or need providers, but any resource type concerned about portability will likely need them.

For instance, there are more than 20 package providers, including providers for package formats like dpkg and rpm along with high-level package managers like apt and yum. A provider's main job is to wrap client-side tools, usually by just calling out to those tools with the right information.

The examples on this page use the apt and dpkg package providers, and the examples used are current as of 0.23.0.

**Note:** Unless you are maintaining existing type and provider code, or the Resource API limitations affect you, you should use the Resource API to create custom resource types, instead of this method.

#### Declaring providers

Providers are always associated with a single resource type, so they are created by calling the `provide` method on that resource type.

The `provide` method takes three arguments plus a block:

- The first argument must be the name of the provider, as a `:symbol`.
- The optional `:parent` argument should be the name of a parent class.
- The optional `:source` argument should be a symbol.
- The block takes no arguments, and implements the behavior of the provider.

There are several kinds of parent classes you can use:

**Base provider**

A provider can inherit from a base provider, which is never used alone and only exists for other providers to inherit from. Use the full name of the class. For example, all package providers have a common parent class:

```
Puppet::Type.type(:package).provide(:dpkg, :pare
 => Puppet::Provider::Package) do
  desc "..."
  ...
end
```

Note the call to the `desc` method; this sets the documentation for this provider, and should include everything necessary for someone to use this provider.

**Another provider of the same resource type**

Providers can also specify another provider as their parent. If it's a provider for the same resource type, you can use the name of that provider as a symbol.

```
Puppet::Type.type(:package).provide(:apt, :parer
 => :dpkg, :source => :dpkg) do
    ...
end
```

Note that we're also specifying that this provider uses the dpkg source; this tells Puppet to deduplicate packages from dpkg and apt, so the same package does not show up in an instance list from each provider type. Puppet defaults to creating a new source for each provider type, so you have to specify when a provider subclass shares a source with its parent class.

**A provider of any resource type**

Providers can also specify a provider of any resource type as their parent. Use the Puppet::Type.type(<NAME>).provider(<NAME>) meth to locate the provider.

```
# my_module/lib/puppet/provider/
glance_api_config/ini_setting.rb
Puppet::Type.type(:glance_api_config).provide(
  :ini_setting,
  # set ini_setting as the parent
 provider
  :parent =>
 Puppet::Type.type(:ini_setting).provider(:ruby)
) do
  # implement section as the first
 part of the namevar
  def section
    resource[:name].split('/',
 2).first
  end
  def setting
    # implement setting as the
 second part of the namevar
    resource[:name].split('/',
 2).last
  end
  # hard code the file path (this
 allows purging)
  def self.file_path
    '/etc/glance/glance-api.conf'
  end
end
```

## Suitability

The first question to ask about a new provider is where it will be functional, which Puppet calls *suitable*. Unsuitable providers cannot be used to do any work. The suitability test is late-binding, meaning that you can have a resource in your configuration that makes a provider suitable.

If you start puppetd or puppet in debug mode, you'll see the results of failed provider suitability tests for the resource types you're using.

Puppet providers include some helpful class-level methods you can use to both document and declare how to determine whether a given provider is suitable. The primary method is commands, which does two things for you:

it declares that this provider requires the named binary, and it sets up class and instance methods with the name provided that call the specified binary. The binary can be fully qualified, in which case that specific path is required, or it can be unqualified, in which case Puppet finds the binary in the shell path and uses that. If the binary cannot be found, then the provider is considered unsuitable. For example, here is the header for the dpkg provider (as of 0.23.0):

```
commands :dpkg => "/usr/bin/dpkg"
commands :dpkg_deb => "/usr/bin/dpkg-deb"
commands :dpkgquery => "/usr/bin/dpkg-query"
```

In addition to looking for binaries, Puppet can compare Facter facts, test for the existence of a file, check for a feature such as a library, or test whether a given value is true or false. For file existence, truth, or false, call the `confine` class method with `exists`, `true`, or `false` as the name of the test and your test as the value:

```
confine :exists => "/etc/debian_release"
confine :true => /^10\.[0-4]/.match(product_version)
confine :false => (Puppet[:ldapuser] == "")
```

To test Facter values, use the name of the fact:

```
confine :operatingsystem => [:debian, :solaris]
confine :puppetversion => "0.23.0"
```

Case doesn't matter in the tests, nor does it matter whether the values are strings or symbols. It also doesn't matter whether you specify an array or a single value — Puppet does an OR on the list of values.

To test a feature, as defined in `lib/puppet/feature/*.rb`, supply the name of the feature. This is preferable to using a `confine :true` statement that calls `Puppet.features` because the expression is evaluated only one time. Puppet enables the provider if the feature becomes available during a run (for example, if a package is installed during the run).

```
confine :feature => :posix
confine :feature => :rrd
```

You can create custom features. They live in `lib/puppet/feature/*.rb` and an example can be found here. These features can be shipped in a similar manner as types and providers are shipped within modules and are pluginsynced.

Using custom features you can delay resource evaluation until the provider becomes suitable. This is a way of informing Puppet that your provider depends on a file being created by Puppet, or a certain fact being set to some value, or it not being set at all.

#### Default providers

Providers are generally meant to be hidden from the users, allowing them to focus on resource specification rather than implementation details, soPuppet does what it can to choose an appropriate default provider for each resource type.

This is generally done by a single provider declaring that it is the default for a given set of facts, using the defaultfor class method. For instance, this is the apt provider's declaration:

```
defaultfor :operatingsystem => :debian
```

The same fact-matching functionality as confinement is used.

Alternatively, you can supply a regular expression (regex) value to match against a fact value. This is useful, for example, for providers that should only be default for a specific range of operating system versions:

```
defaultfor :operatingsystemmajrelease => /^[5-7]$/
```

### How providers interact with resources

Providers do nothing on their own; all of their action is triggered through an associated resource (or, in special cases, from the transaction). Because of this, resource types are essentially free to define their own provider interface if necessary, and providers were initially developed without a clear resource-provider API (mostly because it wasn't clear whether such an API was necessary or what it would look like). At this point, however, there is a default interface between the resource type and the provider.

This interface consists entirely of getter and setter methods. When the resource is retrieving its current state, it iterates across all of its properties and calls the getter method on the provider for that property. For instance, when a `user` resource is having its state retrieved and its `uid` and `shell` properties are being managed, then the resource calls `uid` and `shell` on the provider to figure out what the current state of each of those properties is. This method call is in the `retrieve` method in `Puppet::Property`.

When a resource is being modified, it calls the equivalent setter method for each property on the provider. Again using our user example, if the `uid` was in sync but the `shell` was not, then the resource would call `shell=(value)` with the new shell value.

The transaction is responsible for storing these returned values and deciding which value to send, and it does its work through a `PropertyChange` instance. It calls `sync` on each of the properties, which in turn call the setter by default.

You can override that interface as necessary for your resource type.

All providers must define an `instances` class method that returns a list of provider instances, one for each existing instance of that provider. For example, the dpkg provider should return a provider instance for every package in the dpkg database.

### Provider methods

By default, you have to define all of your getter and setter methods. For simple cases, this is sufficient — you just implement the code that does the work for that property. For the more complicated aspects of provider implementation, Puppet has prefetching, resource methods, and flushing.

#### Prefetching

First, Puppet transactions prefetch provider information by calling `prefetch` on each used provider type. This calls the `instances` method in turn, which returns a list of provider instances with the current resource state already retrieved and stored in a `@property_hash` instance variable. The prefetch method then tries to find any matching resources, and assigns the retrieved providers to found resources. This way you can get information on all of the resources you're managing in just a few method calls, instead of having to call all of the getter methods for every property being managed. Note that it also means that providers are often getting replaced, so you cannot maintain state in a provider.

#### Resource methods

For providers that directly modify the system when a setter method is called, there's no substitute for defining them manually. But for resources that get flushed to disk in one step, such as the `ParsedFile` providers, there is a `mk_resource_methods` class method that creates a getter and setter for each property on the resource. These methods retrieve and set the appropriate value in the `@property_hash` variable.

#### Flushing

Many providers model files or parts of files, so it makes sense to save up all of the writes and do them in one run. Providers that need this functionality can define a flush instance method to do this. The transaction calls this method after all values are synced (which means that the provider should have them all in its `@property_hash` variable) but before refresh is called on the resource (if appropriate).

# Values and data types

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

Strings are the most common and useful data type, but you'll also work with others, including numbers, arrays, and some Puppet-specific data types like resource references.

## Literal data types as values

Although you'll mostly interact with values *of* the various data types, Puppet also includes values like `String` that *represent* data types.

You can use these special values to examine a piece of data or enforce rules. Usually, they act like patterns, similar to a regular expression: given a value and a data type, you can test whether the value *matches* the data type, and then either adjust your code's behavior accordingly, or raise an error if something has gone wrong.

The pages in this section provide details about using each of the data types as a value. For information about the syntax and behavior of literal data types, see Data type syntax. For information about special abstract data types, which you can use to do more sophisticated or permissive type checking, see Abstract data types.

## Puppet's data types

See the following pages to learn more about the syntax, parameters, and usage for each of the data types.

- Strings on page 295
Strings are unstructured text fragments of any length. They're a common and useful data type.
- Numbers on page 304
Numbers in the Puppet language are normal integers and floating point numbers.
- Booleans on page 307
Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.
- Arrays on page 308
Arrays are ordered lists of values. Resource attributes which accept multiple values (including the relationship metaparameters) generally expect those values in an array. Many functions also take arrays, including the iteration functions.
- Hashes on page 311
Hashes map keys to values, maintaining the order of the entries according to insertion order.
- Regular expressions on page 314
A regular expression (sometimes shortened to "regex" or "regexp") is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.
- Sensitive on page 316
Sensitive types in the Puppet language are strings marked as sensitive. The value is displayed in plain text in the catalog and manifest, but is redacted from logs and reports. Because the value is maintained as plain text, you should only use it as an aid to ensure that sensitive values are not inadvertently disclosed.
- Time-related data types on page 316
A `Timespan` defines the length of a duration of time, and a `Timestamp` defines a point in time. For example, "two hours" is a duration that can be represented as a `Timespan`, while "three o'clock in the afternoon UTC on 8 November, 2018" is a point in time that can be represented as a `Timestamp`. Both types can use nanosecond values if it is available on the platform.

- Undef on page 318

Puppet's `undef` value is roughly equivalent to `nil` in Ruby. It represents the absence of a value. If the `strict_variables` setting isn't enabled, variables which have never been declared have a value of `undef`.

- Default on page 319

Puppet's `default` value acts like a keyword in a few specific usages. Less commonly, it can also be used as a value.

- Resource and class references on page 320

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

- Resource types on page 322

Resource types are a special family of data types that behave differently from other data types. They are subtypes of the fairly abstract `Resource` data type. Resource references are a useful subset of this data type family.

- Data type syntax on page 326

Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

- Abstract data types on page 329

If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

## Strings

Strings are unstructured text fragments of any length. They're a common and useful data type.

Strings can interpolate other values, and can use escape sequences to represent characters that are inconvenient or impossible to write literally. You can access substrings of a string by numerical index.

There are four ways to write literal strings in the Puppet language:

- Bare words
- Single-quoted strings
- Double-quoted strings
- Heredocs

Each of these have slightly different behavior around syntax, interpolation features, and escape sequences.

### Bare words

Puppet treats certain bare words — that is, runs of alphanumeric characters without surrounding quotation marks — as single-word strings. Bare word strings are most commonly used with resource attributes that accept a limited number of one-word values.

To be treated as a string, a bare word must:

- Begin with a lower case letter;
- Contain only letters, digits, hyphens (`-`), and underscores (_); and
- Not be a reserved word.

For example, in the following code, `running` is a bare word string:

```
service { "ntp":
  ensure => running, # bare word string
}
```

Unquoted words that begin with upper case letters are interpreted as data types or resource references, not strings.

Bare word strings can't interpolate values and can't use escape sequences.

### Single-quoted strings

Multi-word strings can be surrounded by single quotation marks, `'like this'`.

For example:

```
if $autoupdate {
  notice('autoupdate parameter has been deprecated and replaced with
 package_ensure.  Set this to latest for the same behavior as autoupdate =>
 true.')
}
```

Line breaks within the string are interpreted as literal line breaks.

Single-quoted strings can't interpolate values.

#### Escape sequences

The following escape sequences are available in single-quoted strings:

| Sequence | Result |
|---|---|
| `\\` | Single backslash |
| `\'` | Literal single quotation mark |

Within single quotation marks, if a backslash is followed by any character other than another backslash or a single quotation mark, Puppet treats it as a literal backslash.

To include a literal double backslash use a quadruple backslash.

To include a backslash at the very end of a single-quoted string, use a double backslash instead of a single backslash. For example: `path => 'C:\Program Files(x86)\\'`

**Tip:** A good habit is to always use two backslashes where you want the result to be one backslash. For example, `path => 'C:\\Program Files(x86)\\'`

### Double-quoted strings

Strings can be surrounded by double quotation marks, `"like this"`.

Line breaks within the string are interpreted as literal line breaks. You can also insert line breaks with `\n` (Unix-style) or `\r\n` (Windows-style).

Double-quoted strings can interpolate values. See Interpolation information below.

#### Escape sequences

The following escape sequences are available in double-quoted strings:

| Sequence | Result |
|---|---|
| `\\` | Single backslash |
| `\n` | New line |
| `\r` | Carriage return |
| `\t` | Tab |
| `\s` | Space |
| `\$` | Literal dollar sign (to prevent interpolation) |
| `\uXXXX` | Unicode character number XXXX (a four-digit hexadecimal number) |

| Sequence | Result |
|---|---|
| `\u{XXXXXX}` | Unicode character XXXXXX (a hexadecimal number between two and six digits) |
| `\"` | Literal double quotation mark |
| `\'` | Literal single quotation mark |

Within double quotation marks, if a backslash is followed by any character other than those listed above (that is, a character that is not a recognized escape sequence), Puppet logs a warning: `Warning: Unrecognized escape sequence`, and treats it as a literal backslash.

**Tip:** A good habit is to always use two backslashes where you want the result to be one backslash.

### Heredocs

Heredocs let you quote strings with more control over escaping, interpolation, and formatting. They're especially good for long strings with complicated content.

#### Example

```
$gitconfig = @("GITCONFIG"/L)
    [user]
        name = ${displayname}
        email = ${email}
    [color]
        ui = true
    [alias]
        lg = "log --pretty=format:'%C(yellow)%h%C(reset) %s \
    %C(cyan)%cr%C(reset) %C(blue)%an%C(reset) %C(green)%d%C(reset)' --graph"
        wdiff = diff --word-diff=color --ignore-space-at-eol \
    --word-diff-regex='[[:alnum:]]+|[^[:space:][:alnum:]]+'
    [merge]
        defaultToUpstream = true
    [push]
        default = upstream
    | GITCONFIG

file { "${homedir}/.gitconfig":
  ensure  => file,
  content => $gitconfig,
}
```

#### Syntax

To write a heredoc, you place a heredoc tag in a line of code. This tag acts as a literal string value, but the content of that string is read from the lines that follow it. The string ends when an end marker is reached.

The general form of a heredoc string is:

<table>
<tr>
<td>heredoc tag<br><br><code>@("ENDTEXT"/<X>)</code></td>
<td>You can use a heredoc tag in Puppet code, anywhere a string value is accepted. In the above example, the heredoc tag <code>@("GITCONFIG"/L)</code> completes the line of Puppet code: <code>$gitconfig = </code>.<br><br>A heredoc tag starts with <code>@(</code> and ends with <code>)</code>.<br><br>Between those characters, the heredoc tag contains end text (see below) — text that is used to mark the end of the string. You can optionally surround this end text with double quotation marks to enable interpolation (see below). In the above example, the end text is <code>GITCONFIG</code>.<br><br>It also optionally contains escape switches (see below), which start with a slash <code>/</code>. In the above example, the heredoc tag has the escape switch <code>/L</code>.</td>
</tr>
<tr>
<td>the string<br><br><code>This is the</code><br><code>text that makes</code><br><code>up my string.</code></td>
<td>The content of the string starts on the next line, and can run over multiple lines. If you specified escape switches in the heredoc tag, the string can contain the enabled escape sequences.<br><br>In the above example, the string starts with <code>[user]</code> and ends with <code>default = upstream</code>. It uses the escape sequence <code>\</code> to add cosmetic line breaks, because the heredoc tag contained the <code>/L</code> escape switch.</td>
</tr>
<tr>
<td>end marker<br><br><code>| - ENDTEXT</code></td>
<td>On a line of its own, the end marker consists of:<br><br>• Optional indentation and a pipe character (<code>|</code>) to indicate how much indentation is stripped from the lines of the string (see below).<br>• An optional hyphen character (<code>-</code>), with any amount of space around it, to trim the final line break from the string (see below).<br>• The end text, repeating the same end text used in the heredoc tag, always without quotation marks.<br><br>In the above example, the end marker is <code>| GITCONFIG</code>.</td>
</tr>
</table>

If a line of code includes more than one heredoc tag, Puppet reads all of those heredocs in order: the first one begins on the following line and continue until its end marker, the second one begins on the line immediately after the first end marker, and so on. Puppet won't start evaluating additional lines of Puppet code until it reaches the end marker for the final heredoc tag from the original line.

### End text

The heredoc tag contains piece of text called the *end text*. When Puppet reaches a line that contains only that end text (plus optional formatting control), the string ends.

Both occurrences of the end text must match exactly, with the same capitalization and internal spacing.

End text can be any run of text that doesn't include line breaks, colons, slashes, or parentheses. It can include spaces, and can be mixed case. The following are all valid end text:

- `EOT`
- `...end...end...`

- `Verse 8 of The Raven`

**Tip:** To help with code readability, make your end text stand out from the content of the heredoc string, for example, by making it all uppercase.

### Enabling interpolation

By default, heredocs do not allow you to interpolate values into the string content. You can enable interpolation by double-quoting the end text in the opening heredoc tag. That is:

- An opening tag like `@(EOT)` won't allow interpolation.
- An opening tag like `@("EOT")` allows interpolation.

**Note:** If you enable interpolation, but the string has a dollar character (`$`) that you want to be literal, not interpolated, you must enable the literal dollar sign escape switch (`/$`) in the heredoc tag: `@("EOT"/$)`. Then use the escape sequence `\$` to specify the literal dollar sign in the string. See the information on enabling escape sequences, below, for more details.

### Enabling escape sequences

By default, heredocs have no escape sequences and every character is literal (except interpolated expressions, if enabled). To enable escape sequences, add switches to the heredoc tag.

To enable individual escape sequences, add a slash (`/`) and one or more switches. For example, to enable an escape sequence for dollar signs (`\$`) and new lines (`\n`), add `/$n` to the heredoc tag :

```
@("EOT"/$n)
```

To enable all escape sequences, add a slash and no switches:

```
@("EOT"/)
```

Use the following switches to enable escape sequences:

| Switch to put in the heredoc tag | Escape sequence to use in the heredoc string | Result in the string value |
|---|---|---|
| (automatic) | `\\` | Single backslash. This switch is enabled when any other escape sequence is enabled. |
| `n` | `\n` | New line |
| `r` | `\r` | Carriage return |
| `t` | `\t` | Tab |
| `s` | `\s` | Space |
| `$` | `\$` | Literal dollar sign (to prevent interpolation) |
| `u` | `\uXXXX` or `\u{XXXXXX}` | Unicode character number `XXXX` (a four-digit hexadecimal number) or `XXXXXX` (a two- to six-digit hexadecimal number) |
| `L` | \<New line or carriage return> | Nothing. This lets you put line breaks in the heredoc source code that won't appear in the string value. |

**Note:** A backslash that isn't part of an escape sequence is treated as a literal backslash. Unlike in double-quoted strings, this won't log a warning.

**Tip:** If a heredoc has escapes enabled, and includes several literal backslashes in a row, make sure each literal backslash is represented by the `\\` escape sequence. So, for example, If you want the result to include a double backslash, use four backslashes.

### Enabling syntax checking

To enable syntax checking of heredoc text, add the name of the syntax to the heredoc tag. For example:

```
@(END:pp)
@(END:epp)
@(END:json)
```

If Puppet has a syntax checker for the given syntax, it validates the heredoc text, but only if the heredoc is static text and does not contain any interpolations. If Puppet has no checker available for the given syntax, it silently ignores the syntax tag.

Syntax checking in heredocs is useful for validating syntax earlier, avoiding later failure.

By default, heredocs are treated as text unless otherwise specified in the heredoc tag.

### Stripping indentation

To make your code easier to read, you can indent the content of a heredoc to separate it from the surrounding code. To strip this indentation from the resulting string value, put the same amount of indentation in front of the end marker and use a pipe character (`|`) to indicate the position of the first “real” character on each line.

```
$mytext = @(EOT)
    This block of text is
    visibly separated from
    everything around it.
    | EOT
```

If a line has less indentation than you’ve indicated with the pipe, Puppet strips any spaces it can without deleting non-space characters.

If a line has more indentation than you’ve indicated with the pipe, the excess spaces are included in the final string value.

**Important:** Indentation can include tab characters, but Puppet won’t convert tabs to spaces, so make sure you use the exact same sequence of space and tab characters on each line.

### Suppressing literal line breaks

If you enable the `L` escape switch, you can end a line with a backslash (`\`) to exclude the following line break from the string value. This lets you break up long lines in your source code without adding unwanted literal line breaks to the resulting string value.

For example, Puppet would read this as a single line:

```
lg = "log --pretty=format:'%C(yellow)%h%C(reset) %s \
%C(cyan)%cr%C(reset) %C(blue)%an%C(reset) %C(green)%d%C(reset)' --graph"
```

### Suppressing the final line break

By default, heredocs end with a trailing line break, but you can exclude this line break from the final string. To suppress it, add a hyphen (`-`) to the end marker, before the end text, but after the indentation pipe if you used one. This works even if you don’t have the `L` escape switch enabled.

For example, Puppet would read this as a string with no line break at the end:

```
$mytext = @("EOT")
```

```
    This is too inconvenient for ${double} or ${single} quotes, but must be
one line.
    |-EOT
```

#### Interpolation

Interpolation allows strings to contain expressions, which can be replaced with their values. You can interpolate any expression that resolves to a value, except for statement-style function calls. You can interpolate expressions in double-quoted strings, and in heredocs with interpolation enabled.

To interpolate an expression, start with a dollar sign and wrap the expression in curly braces, as in: `"String content ${<EXPRESSION>} more content"`.

The dollar sign doesn't have to have a space in front of it. It can be placed directly after any other character, or at the beginning of the string.

An interpolated expression can include quote marks that would end the string if they occurred outside the interpolation token. For example: `"<VirtualHost *:${hiera("http_port")}>"`.

#### Preventing interpolation

If you want a string to include a literal sequence that looks like an interpolation token, but you don't want Puppet to try to evaluate it, use a quoting syntax that disables interpolation (single quotes or a non-interpolating heredoc), or escape the dollar sign with `\$`.

#### Short forms for variable interpolation

The most common thing to interpolate into a string is the value of a variable. To make this easier, Puppet has some shorter forms of the interpolation syntax:

**`$myvariable`**

A variable reference (without curly braces) can be replaced with that variable's value. This also works with qualified variable names like `$myclass::myvariable`.

Because this syntax doesn't have an explicit stopping point (like a closing curly brace), Puppet assumes the variable name is everything between the dollar sign and the first character that couldn't legally be part of a variable name. (Or the end of the string, if that comes first.)

This means you can't use this style of interpolation when a value must run up against some other word-like text. And even in some cases where you can use this style, the following style can be clearer.

**`${myvariable}`**

A dollar sign followed by a variable name in curly braces can be replaced with that variable's value. This also works with qualified variable names like `${myclass::myvariable}`.

With this syntax, you can follow a variable name with any combination of chained function calls or hash access / array access / substring access expressions. For example:

```
"Using interface
 ${::interfaces.split(',')[3]} for
 broadcast"
```

However, this doesn't work if the variable's name overlaps with a language keyword. For example, if you had a variable called `$inherits`, you would have to use normal-style interpolation:

```
"Inheriting ${$inherits.upcase}."
```

### Conversion of interpolated values

Puppet converts the value of any interpolated expression to a string using these rules:

| Data type | Conversion |
| --- | --- |
| String | The contents of the string, with any quoting syntax removed. |
| Undef | An empty string. |
| Boolean | The string `'true'` or `'false'`. |
| Number | The number in decimal notation (base 10). For floats, the value can vary on different platforms. Use the `sprintf` function for more precise formatting. |
| Array | A pair of square brackets (`[` and `]`) containing the array's elements, separated by a comma and a space (`, `), with no trailing comma. Each element is converted to a string using these rules. |
| Hash | A pair of curly braces (`{` and `}`) containing a `<KEY> => <VALUE>` string for each key-value pair, separated by a comma and a space (`, `), with no trailing comma. Each key and value is converted to a string using these rules. |
| Regular expression | A stringified regular expression. |
| Resource reference or data type | The value as a string. |

### Line breaks

Quoted strings can continue over multiple lines, and line breaks are preserved as a literal part of the string. Heredocs let you suppress these line breaks if you use the `L` escape switch.

Puppet does not attempt to convert line breaks, so whatever type of line break is used in the file (LF for *nix or CRLF for Windows) is preserved. Use escape sequences to insert specific types of line breaks into strings:

- To insert a CRLF in a manifest file that uses *nix line endings, use the `\r\n` escape sequences in a double-quoted string, or a heredoc with those escapes enabled.
- To insert an LF in a manifest that uses Windows line endings, use the `\n` escape sequence in a double-quoted string, or a heredoc with that escape enabled.

### Encoding

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

However, all strings should be valid UTF-8. Future versions of Puppet might impose restrictions on string encoding, and using only UTF-8 protects you in this event. Also, PuppetDB removes invalid UTF-8 characters when storing catalogs.

### Accessing substrings

Access substrings of a string by specifying a numerical index inside square brackets. The index consists of one integer, optionally followed by a comma and a second integer, for example `$string[3]` or `$string[3,10]`.

The first number of the index is the start position. Positive numbers count from the start of the string, starting at `0`. Negative numbers count back from the end of the string, starting at `-1`.

The second number of the index is the stop position. Positive numbers are lengths, counting forward from the start position. Negative numbers are absolute positions, counting back from the end of the string (starting at `-1`). If the second number is omitted, it defaults to `1` (resolving to a single character).

Examples:

```
$mystring = 'abcdef'
notice( $mystring[0] )    # resolves to 'a'
notice( $mystring[0,2] )  # resolves to 'ab'
notice( $mystring[1,2] )  # resolves to 'bc'
notice( $mystring[1,-2] ) # resolves to 'bcde'
notice( $mystring[-3,2] ) # resolves to 'de'
```

Text outside the actual range of the string is treated as an infinite amount of empty string:

```
$mystring = 'abcdef'
notice( $mystring[10] )    # resolves to ''
notice( $mystring[3,10] )  # resolves to 'def'
notice( $mystring[-10,2] ) # resolves to ''
notice( $mystring[-10,6] ) # resolves to 'ab'
```

#### The `String` data type

The data type of strings is `String`. By default, `String` matches strings of any length. You can use parameters to restrict which values `String` matches.

#### Parameters

The full signature for `String` is:

```
String[<MIN LENGTH>, <MAX LENGTH>]
```

These parameters are optional. They must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Minimum length | Integer | 0 | The minimum number of (Unicode) characters in the string. This parameter accepts the special value `default`, which uses its default value. |
| 2 | Maximum length | Integer | infinite | The maximum number of (Unicode) characters in the string. This parameter accepts the special value `default`, which uses its default value. |

Examples:

| | |
|---|---|
| **`String`** | Matches a string of any length. |
| **`String[6]`** | Matches a string with at least six characters. |

| | |
|---|---|
| **String[6,8]** | Matches a string with at least six and at most eight characters. |

### Related information

Data type syntax on page 326
Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Abstract data types on page 329
If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

## Numbers

Numbers in the Puppet language are normal integers and floating point numbers.

You can work with numbers using arithmetic operators.

Numbers are written without quotation marks, and can consist only of:

- Digits.
- An optional negative sign (`-`). This is actually the unary negation operator rather than part of the number. Explicit positive signs (`+`) aren't allowed.
- An optional decimal point, which results in a floating point value.
- An optional `e` or `E` for scientific notation of floating point values.
- An `0` prefix for octal base, or `0x` or `0X` prefix hexidecimal base.

### Integers

Integers are numbers without decimal points.

If you divide two integers, the result is not a float. Instead, Puppet truncates the remainder. For example:

```
$my_number = 2 / 3    # evaluates to 0
$your_number = 5 / 3  # evaluates to 1
```

### Floating point numbers

Floating point numbers ("floats") are numbers that include a fractional value after a decimal point, including a fractional value of zero, as in `2.0`.

If an expression includes both integer and float values, the result is a float:

```
$some_number = 8 * -7.992           # evaluates to -63.936
$another_number = $some_number / 4  # evaluates to -15.984
```

Floating point numbers between -1 and 1 cannot start with a bare decimal point. They must have a zero before the decimal point:

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

You can express floating point numbers in scientific notation: append `e` or `E`, plus an exponent, and the preceding number is multiplied by 10 to the power of that exponent. Numbers in scientific notation are always floats:

```
$product = 8 * 3e5  # evaluates to 2400000.0
```

### Octal and hexadecimal integers

Integer values can be expressed in decimal notation (base 10), octal notation (base 8), and hexadecimal notation (base 16).

Decimal (base 10) integers (other than `0`) must not start with a `0`.

Octal (base 8) integers have a prefix of `0` (zero), followed by octal digits `0` to `7`.

Hexadecimal (base 16) integers have a prefix of 0x or 0X, followed by hexadecimal digits `0` to `9`, `a` to `f`, or `A` to `F`.

Floats can't be expressed in octal or hexadecimal.

Examples:

```
# octal
$value = 0777   # evaluates to decimal 511
$value = 0789   # Error, invalid octal
$value = 0777.3 # Error, invalid octal

# hexadecimal
$value = 0x777 # evaluates to decimal 1911
$value = 0xdef # evaluates to decimal 3567
$value = 0Xdef # same as above
$value = 0xDEF # same as above
$value = 0xLMN # Error, invalid hex
```

### Converting numbers to strings

Numbers are automatically converted to strings when interpolated into a string. The automatic conversion uses decimal (base 10) notation.

To convert numbers to non-decimal string representations, use the `sprintf` function.

### Converting strings to numbers

Arithmetic operators in an expression automatically converts strings to numbers, but in all other contexts (for example, resource attributes or function arguments), Puppet won’t automatically convert strings to numbers

To convert a string to a number, add zero (`0`) to it. For example:

```
$mystring = "85"
$mynum = 0 + $mystring  # mynum is an integer with value 85
```

To extract numbers from strings, use the `scanf` function. This function handles surrounding non-numerical text.

### The `Integer` data type

The data type of integers is `Integer`. By default, `Integer` matches whole numbers of any size, within the limits of available memory. You can use parameters to restrict which values `Integer` matches.

### Parameters

The full signature for `Integer` is:

```
Integer[<MIN VALUE>, <MAX VALUE>]
```

These parameters are optional. They must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

| Position | Parameter | Data type | Default | Description |
| --- | --- | --- | --- | --- |
| 1 | Minimum value | `Integer` | negative infinity | The minimum value for the integer. This parameter accepts the special value `default`, which uses its default value. |
| 2 | Maximum value | `Integer` | infinity | The maximum value for the integer. This parameter accepts the special value `default`, which uses its default value. |

Practically speaking, the integer size limit is the range of a 64-bit signed integer (−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807), which is the maximum size that can roundtrip safely between the components in the Puppet ecosystem.

Examples:

| | |
| --- | --- |
| **`Integer`** | Matches any integer. |
| **`Integer[0]`** | Matches any integer greater than or equal to 0. |
| **`Integer[default, 0]`** | Matches any integer less than or equal to 0. |
| **`Integer[2, 8]`** | Matches any integer from 2 to 8, inclusive. |

#### The `Float` data type

The data type of floating point numbers is `Float`. By default, `Float` matches floating point numbers within the limitations of Ruby's Float class. Practically speaking, this means a 64-bit double precision floating point value. You can use parameters to restrict which values `Float` matches.

#### Parameters

The full signature for `Float` is:

```
Float[<MIN VALUE>, <MAX VALUE>]
```

These parameters are optional. They must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

| Position | Parameter | Data type | Default | Description |
| --- | --- | --- | --- | --- |
| 1 | Minimum value | `Float` | negative infinity | The minimum value for the float. This parameter accepts the special value `default`, which uses its default value. |

| Position | Parameter | Data type | Default | Description |
|---|---|---|---|---|
| 2 | Maximum value | `Float` | infinity | The maximum value for the float. This parameter accepts the special value `default`, which uses its default value. |

Examples:

| | |
|---|---|
| **`Float`** | Matches any floating point number. |
| **`Float[1.6]`** | Matches any floating point number greater than or equal to 1.6. |
| **`Float[1.6, 3.501, 0]`** | Matches any floating point number from 1.6 to 3.501, inclusive. |

For more information about Float see Ruby's Float class docs.

### The Numeric data type

The data type of all numbers, both integer and floating point, is `Numeric`. It matches any integer or floating point number, and takes no parameters.

`Numeric` is equivalent to `Variant[Integer, Float]`. If you need to set size limits but still accept both integers and floats, you can use the abstract type `Variant` to construct an appropriate data type. For example:

```
Variant[Integer[-3,3], Float[-3.0,3.0]]
```

### Related information

Data type syntax on page 326
Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Abstract data types on page 329
If you’re using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

## Booleans

Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.

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

### Automatic conversion to boolean

If a non-boolean value is used where a boolean is required:

- The `undef` value is converted to boolean `false`.
- All other values are converted to boolean `true`.

Notably, this means the string values `""` (a zero-length string) and `"false"` (in quotation marks) both resolve to `true`.

To convert values to booleans with more permissive rules (for example, `0` to `false`, or `"false"` to `false`), use the `str2bool` and `num2bool` functions in the `puppetlabs-stdlib` module.

### The `Boolean` data type

The data type of boolean values is `Boolean`.

It matches only the values `true` or `false`, and accepts no parameters.

You can use abstract types to match values that might be boolean or might have some other value. For example, `Optional[Boolean]` matches `true`, `false`, or `undef`. `Variant[Boolean, Enum["true", "false"]]` matches stringified booleans as well as true booleans.

### Related information

Data type syntax on page 326
Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Abstract data types on page 329
If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

## Arrays

Arrays are ordered lists of values. Resource attributes which accept multiple values (including the relationship metaparameters) generally expect those values in an array. Many functions also take arrays, including the iteration functions.

Arrays are written as comma-separated lists of values 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 values in an array can be any data type.

### Accessing items in an array

You can access items in an array by their numerical index, counting from zero. Square brackets are used for accessing. For example:

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

This manifest would log `two` as a notice. The value of `$my_array[0]` would be `one`, because indexes count from zero.

The opening square bracket must not be preceded by white space:

```
$my_array = [ 'one', 'two', 'three', 'four', 'five' ]
notice( $my_array[2] )  # ok
notice( $my_array [2] ) # syntax error
```

Nested arrays and hashes can be accessed by chaining indexes:

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

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

Arrays support negative indexes, counting backward from the last element, with `-1` being the last element of the array:

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

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

If you try to access an element beyond the bounds of the array, its value is `undef`:

```
$my_array = [ 'one', 'two', 'three', 'four', 'five' ]
$cool_value = $my_array[6] # value is undef
```

When testing with a regular expression whether an `Array[<TYPE>]` data type matches a given array, empty arrays match if the type accepts zero-length arrays.

```
$my_array = []
if $my_array =~ Array[String] {
  notice( 'my_array' )
}
```

This manifest would log `my_array` as a notice, because the expression matches the empty array.

### Accessing sections of an array

You can access sections of an array by numerical index. Like accessing individual items, accessing sections uses square brackets, but it uses two indexes, separated by a comma. For example: `$array[3,10]`.

The result of an array section is always another array.

The first number of the index is the start position. Positive numbers count forward from the start of the array, starting at zero. Negative numbers count backward from the end of the array, starting at `-1`.

The second number of the index is the stop position. Positive numbers are lengths, counting forward from the start position. Negative numbers are absolute positions, counting backward from the end of the array starting at `-1`.

For example:

```
$my_array = [ 'one', 'two', 'three', 'four', 'five' ]
notice( $my_array[2,1] )  # evaluates to ['three']
notice( $my_array[2,2] )  # evaluates to ['three', 'four']
notice( $my_array[2,-1] ) # evaluates to ['three', 'four', 'five']
notice( $my_array[-2,1] ) # evaluates to ['four']
```

### Array operators

You can use operators with arrays to create new arrays:

- append with `<<`
- concatenate with `+`
- remove elements with `-`

or to convert arrays to comma-separated lists with `*` (splat). See the Array operators section of Expressions for more information.

### Additional functions for arrays

The `puppetlabs-stdlib` module contains useful functions for working with arrays, including:

| | | |
|---|---|---|
| `delete` | `join` | `size` |

| | | |
|---|---|---|
| delete_at | member | sort |
| flatten | prefix | unique |
| grep | range | validate_array |
| hash | reverse | values_at |
| is_array | shuffle | zip |

#### Related information

Regular expressions on page 314
A regular expression (sometimes shortened to “regex” or “regexp”) is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

Undef on page 318
Puppet's `undef` value is roughly equivalent to `nil` in Ruby. It represents the absence of a value. If the `strict_variables` setting isn’t enabled, variables which have never been declared have a value of `undef`.

### The `Array` data type

The data type of arrays is `Array`. By default, `Array` matches arrays of any length, provided all values in the array match the abstract data type `Data`. You can use parameters to restrict which values `Array` matches.

#### Parameters

The full signature for `Array` is:

```
Array[<CONTENT TYPE>, <MIN SIZE>, <MAX SIZE>]
```

These parameters are optional. They must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Content type | `Type` | `Data` | The kind of values the array contains. You can specify only one data type per array, and every value in the array must match that type. Use an abstract type to allow multiple data types. If the order of elements matters, use the `Tuple` type instead of `Array`. |
| 2 | Minimum size | `Integer` | 0 | The minimum number of elements in the array. This parameter accepts the special value `default`, which uses its default value. |

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 3 | Maximum size | `Integer` | infinite | The maximum number of elements in the array. This parameter accepts the special value `default`, which uses its default value. |

Examples:

| | |
|---|---|
| **`Array`** | Matches an array of any length. All elements in the array must match `Data`. |
| **`Array[String]`** | Matches an array of any size that contains only strings. |
| **`Array[Integer, 6]`** | Matches an array containing at least six integers. |
| **`Array[Float, 6, 12]`** | Matches an array containing at least six and at most 12 floating-point numbers. |
| **`Array[Variant[String, Integer]]`** | Matches an array of any size that contains only strings or integers, or both. |
| **`Array[Any, 2]`** | Matches an array containing at least two elements, allowing any data type (including `Type` and `Resource`). |

The abstract `Tuple` data type lets you specify data types for every element in an array, in order. Abstract types, such as `Variant` and `Enum`, are useful when specifying content types for arrays that include multiple kinds of data.

### Related information

Abstract data types on page 329

If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

## Hashes

Hashes map keys to values, maintaining the order of the entries according to insertion order.

### Syntax

Hashes are written as a pair of curly braces `{}` containing any number of key-value pairs. A key is separated from its value by an arrow (sometimes called a 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 can be any data type, but generally, you should use only strings. Put quotation marks around keys that are strings. Don't assign a hash with non-string keys to a resource attribute or class parameter, because Puppet cannot serialize non-string hash keys into the catalog.

```
{ 'key1' => ['val1','val2'],
  'key2 => {  'key3' =>  'val3',  },
  'key4' => true,
  'key5' => 12345,
```

```
}
```

### Accessing values

Access hash members with their key inside square brackets:

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

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

If you try to access a nonexistent key from a hash, its value is `undef`:

```
$cool_value = $myhash[absent_key] # Value is undef
```

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.

### Merging hashes

When hashes are merged (using the `+` operator), the keys in the constructed hash have the same order as in the original hashes, with the left hash keys ordered first, followed by any keys that appeared only in the hash on the right side of the merge.

Where a key exists in both hashes, the merged hash uses the value of the key in the hash to the right of the `+` operator. For example:

```
$values = {'a' => 'a', 'b' => 'b'}
$overrides = {'a' => 'overridden'}
$result = $values + $overrides
notice($result)
-> {'a' => 'overridden', 'b' => 'b'}
```

### Additional functions for hashes

The `puppetlabs-stdlib` module contains useful functions for working with hashes, including:

| | | |
|---|---|---|
| `has_key` | `keys` | `validate_hash` |
| `is_hash` | `merge` | `values` |

### The `Hash` data type

The data type of hashes is `Hash`. By default, `Hash` matches hashes of any size, as long as their keys match the abstract type `Scalar` and their values match the abstract type `Data`. You can use parameters to restrict which values `Hash` matches.

#### Parameters

The full signature for `Hash` is:

```
Hash[<KEY TYPE>, <VALUE TYPE>, <MIN SIZE>, <MAX SIZE>]
```

These parameters are optional. You must specify both key type and value type if you're going to specify one of them. The parameters must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Key type | `Type` | `Scalar` | What kinds of values can be used as keys. If you specify a key type, a value type is mandatory. |
| 2 | Value type | `Type` | `Data` | What kinds of values can be used as values. |
| 3 | Minimum size | `Integer` | `0` | The minimum number of key-value pairs in the hash. This parameter accepts the special value `default`, which uses its default value. |
| 4 | Maximum size | `Integer` | infinite | The maximum number of key-value pairs in the hash. This parameter accepts the special value `default`, which uses its default value. |

Examples:

| | |
|---|---|
| **`Hash`** | Matches a hash of any length; all keys must match `Scalar` and any values must match `Data`. |
| **`Hash[Integer, String]`** | Matches a hash that uses integers for keys and strings for values. |
| **`Hash[Integer, String, 1]`** | Same as previous, and requires a non-empty hash. |
| **`Hash[Integer, String, 1, 8]`** | Same as previous, and with a maximum size of eight key-value pairs. |

The abstract `Struct` data type lets you specify the exact keys allowed in a hash, as well as what value types are allowed for each key.

Other abstract types, particularly `Variant` and `Enum`, are useful when specifying a value type for hashes that include multiple kinds of data.

#### Related information

Data type syntax on page 326
Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Abstract data types on page 329
If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

## Regular expressions

A regular expression (sometimes shortened to "regex" or "regexp") is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

You can use regular expression values with the `=~` and `!~` match operators, case statements and selectors, node definitions, and functions like `regsubst` for editing strings, or `match` for capturing and extracting substrings. Regular expressions act like any other value, and can be assigned to variables and used in function arguments.

### Syntax

Regular expressions are written as patterns enclosed within forward slashes. Unlike in Ruby, you cannot specify options or encodings after the final slash, like `/node .*/m`.

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

Puppet uses Ruby's standard regular expression implementation to match patterns. Other forms of regular expression quoting, like Ruby's `%r{^www(\d+)\.}`, are not allowed. You cannot interpolate variables or expressions into regex values.

If you are matching against a string that contains newlines, use `\A` and `\z` instead of `^` and `$`, which match the beginning and end of a line. This is a common mistake that can cause your regexp to unintentionally match multiline text.

Some places in the language accept both real regex values and stringified regexes — that is, the same pattern quoted as a string instead of surrounded by slashes.

### Regular expression options

Regular expresions in Puppet cannot have options or encodings appended after the final slash. However, you can 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 available:

| | |
|---|---|
| **i** | Ignore case. |
| **m** | Treat a new line as a character matched by `.` |

| | |
|---|---|
| **x** | Ignore whitespace and comments in the pattern. |

### Regular expression capture variables

Within conditional statements and node definitions, substrings withing parentheses `()` in a regular expression are available as numbered variables inside the associated code section. The first is `$1`, the second is `$2`, and so on. The entire match is 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.
- You can't manually assign values to a variable with only digits in its name; they can only be set by pattern matching.
- 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.

### The Regexp data type

The data type of regular expressions is `Regexp`. By default, `Regexp` matches any regular expression value. If you are looking for a type that matches strings which match arbitrary regular expressions, see the Pattern type. You can use parameters to restrict which values `Regexp` matches.

#### Parameters

The full signature for `Regexp` is:

```
Regexp[<SPECIFIC REGULAR EXPRESSION>]
```

The parameter is optional.

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Specific regular expression | `Regexp` | none | If specified, this results in a data type that only matches one specific regular expression value. Specifying a parameter here doesn't have a practical use. |

Examples:

| | |
|---|---|
| **`Regexp`** | Matches any regular expression. |
| **`Regexp[/<regex>/]`** | Matches the regular expression `/<regex>/` only. |

`Regexp` matches only literal regular expression values. Don't confuse it with the abstract `Pattern` data type, which uses a regular expression to match a limited set of `String` values.

#### Related information

Data type syntax on page 326
Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Abstract data types on page 329

If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

## Sensitive

Sensitive types in the Puppet language are strings marked as sensitive. The value is displayed in plain text in the catalog and manifest, but is redacted from logs and reports. Because the value is maintained as plain text, you should only use it as an aid to ensure that sensitive values are not inadvertently disclosed.

The `Sensitive` type can be written as `Sensitive.new(val)`, or the short form `Sensitive(val)`.

### Parameters

The full signature for `Sensitive` is:

```
Sensitive.new([<STRING VALUE>])
```

The `Sensitive` type is parameterized, but the parameterized type (the type of the value it contains) only retains the basic type. Sensitive information about the length or details about the contained data value can otherwise be leaked.

It is therefore not possible to have detailed data types and expect that the data type match. For example, `Sensitive[Enum[red, blue, green]]` fails if a value of `Sensitive('red')` is given. When a sensitive type is used, the type parameter must be generic; in this example a `Sensitive[String]` instead would match `Sensitive('red')`.

Consider, for example, if you assign a sensitive value and call `notice`:

```
$secret = Sensitive('myPassword')
notice($secret)
```

The example manifest would log the following notice:

```
Notice: Scope(Class[main]): Sensitive [value redacted]
```

To gain access to the original data, use the `unwrap` function:

```
$secret = Sensitive('myPassword')
$processed = $secret.unwrap
notice $processed
```

Use `Sensitive` and `unwrap` only as an aid for logs and reports. The data is not encrypted.

## Time-related data types

A `Timespan` defines the length of a duration of time, and a `Timestamp` defines a point in time. For example, "two hours" is a duration that can be represented as a `Timespan`, while "three o'clock in the afternoon UTC on 8 November, 2018" is a point in time that can be represented as a `Timestamp`. Both types can use nanosecond values if it is available on the platform.

### The `Timespan` data type

A `Timespan` value represents a duration of time. The `Timespan` data type matches a specified range of durations and includes all values within the given range. The `default` represents a positive or negative infinite duration. A `Timespan` value can be specified with strings or numbers in various forms. The type takes up to two parameters.

### Parameters

The full signature for `Timespan` is:

```
Timespan[ (<TIMESPAN START OR LENGTH>, (<TIMESPAN END>)) ]
```

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Timespan start or length | `String`, `Float`, `Integer`, or `default` | `default` (negative infinity in a span) | If only one parameter is passed, it is the length of the timespan. If two parameters are passed, this is the start or `from` value in a time range. |
| 2 | Timespan end | `String`, `Float`, `Integer`, or `default` | `default` (positive infinity) or none if only one value passed. | The end or `to` value in a time range. |

`Timespan` values are interpreted depending on their format:

- A String in the format `D-HH:MM:SS` represents a duration of `D` days, `HH` hours, `MM` minutes, and `SS` seconds.
- An Integer or Float represents a duration in seconds.

A `Timespan` defined as a range (two parameters) matches any `Timespan` durations that can fit within that range. If either end of a range is defined as `default` (infinity), it is an *open range*, while any other range is a *closed range*. The range is inclusive.

In other words, `Timespan[2]` matches a duration of two seconds, but `Timespan[0, 2]` can match *any* `Timespan` from zero to two seconds, inclusive.

The `Timespan` type is not enumerable.

For information about converting values of other types to `Timespan` using the `new` function, or for converting a `Timespan` to a `String` using `strftime`, see the function reference documentation.

Examples:

| | |
|---|---|
| **`Timespan[2]`** | Matches a `Timespan` value of 2 seconds. |
| **`Timespan[77.3]`** | Matches a `Timespan` value of 1 minute, 17 seconds, and 300 milliseconds (77.3 seconds). |
| **`Timespan['1-00:00:00', '2-00:00:00']`** | Matches a closed range of `Timespan` values between 1 and 2 days. |

### The `Timestamp` data type

A `Timestamp` value represents a specific point in time. The `Timestamp` data type can be one single point or any point within a given range, depending on the number of specified parameters. `Timestamp` values that include a `default` parameter represents an infinite range of either positive or negative Timestamps. A `Timestamp` value can be specified with strings or numbers in various forms.

#### Parameters

The full signature for `Timestamp` is:

```
Timestamp[ (<TIMESTAMP VALUE>, (<RANGE LIMIT>)) ]
```

The type takes up to two parameters, and defaults to an infinite range to the past and future. A `Timestamp` with one parameter represents a single point in time, while two parameters represent a range of `Timestamp`s, with the first parameter being the `from` value and the second being the `to` value.

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Timestamp value | `String`, `Float`, `Integer`, or `default` | `default` (negative infinity in a range) | Point in time if passed alone, or `from` value in a range if passed with a second parameter. |
| 2 | Range limit | `String`, `Float`, `Integer`, or `default` | `default` (positive infinity), or none if only one value is passed | The `to` value in a range. |

A `Timestamp` that is defined as a single point in time (one parameter) matches exactly that point.

A `Timestamp` that is defined as a range (two parameters) matches any point in time within that range. If either end of a range is defined as `default` (infinity), it is an *open range*, while any other range is a *closed range*. The range is inclusive.

So, `Timestamp['2000-01-01T00:00:00.000']` matches 0:00 UTC on 1 January, 2000, while `Timestamp['2000-01-01T00:00:00.000', '2001-01-01T00:00:00.000]` matches `Timestamp` values from that point in time to a point in time one year later, inclusive.

`Timestamp` values are interpreted depending on their format.

For information about converting values of other types to `Timestamp` using the `new` function, or for converting a `Timespan` to a `String` using the `strftime` function, see the function reference documentation.

Examples:

| | |
|---|---|
| **`Timestamp['2000-01-01T00:00:00.000', default]`** | Matches an open range of `Timestamps` from the start of the 21st century to an infinite point in the future. |
| **`Timestamp['2012-10-10']`** | Matches the exact `Timestamp` 2012-10-10T00:00:00.0 UTC. |
| **`Timestamp[default, 1433116800]`** | Matches an open range of `Timestamps` from an infinite point in the past to 2015-06-01T00:00:00 UTC, here expressed as seconds since the Unix epoch. |
| **`Timestamp['2010-01-01', '2015-12-31T23:59:59.999999999']`** | Matches a closed range of `Timestamps` between the start of 2010 and the end of 2015. |

## Undef

Puppet's `undef` value is roughly equivalent to `nil` in Ruby. It represents the absence of a value. If the `strict_variables` setting isn't enabled, variables which have never been declared have a value of `undef`.

The `undef` value is useful for testing whether a variable has been set. Also, you can use it to un-set resource attributes that have inherited values from a resource default, causing the attribute to be unmanaged.

The only value in the `Undef` data type is the bare word `undef`.

### Conversion

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

When interpolated into a string, `undef` is converted to the empty string.

### The `Undef` data type

The data type of `undef` is `Undef`. It matches only the value `undef`, and takes no parameters.

Several abstract data types can match the `undef` value:

- The `Data` type matches `undef` in addition to several other data types.
- The `Any` type matches any value, including `undef`.
- The `Optional` type wraps one other data type, and returns a type that matches `undef` in addition to that type.
- The `Variant` type can accept the `Undef` type as a parameter, which makes the resulting data type match `undef`.
- The `NotUndef` type matches any value except `undef`.

#### Related information

Data type syntax on page 326
Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Abstract data types on page 329
If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

## Default

Puppet's `default` value acts like a keyword in a few specific usages. Less commonly, it can also be used as a value.

### Syntax

The only value in the default data type is the bare word `default`.

### Usage with cases and selectors

In case statements and selector expressions, you can use `default` as a case. Puppet attempts to match a `default` case last, after it has tried to match against every other case.

### Usage with per-block resource defaults

You can use `default` as the title in a resource declaration to invoke a particular behavior. Instead of creating a resource and adding it to the catalog, the `default` resource sets fallback attributes that can be used by any other resource in the same resource expression.

In the following example, all of the resources in the block inherits attributes from `default` unless they specifically override them:

```
file {
  default:
    ensure => file,
    mode   => '0600',
    owner  => 'root',
    group  => 'root',
  ;
  '/etc/ssh_host_dsa_key':
  ;
  '/etc/ssh_host_key':
  ;
  '/etc/ssh_host_dsa_key.pub':
    mode => '0644',
  ;
  '/etc/ssh_host_key.pub':
    mode => '0644',
  ;
}
```

For more information, see Resources (Advanced).

#### Usage as parameters of data types

Several data types take parameters that have default values. In some cases, like minimum and maximum sizes, the default value can be difficult or impossible to refer to using the available literal values in the Puppet language. For example, the default value of the `String` type's maximum length parameter is infinity, which can't be represented in the Puppet language.

These parameters let you provide a value of `default` to indicate that you want the default value.

#### Other `default` usage

You can use the value `default` anywhere you aren't prohibited from using it. In these cases, it generally won't have any special meaning.

There are a few reasons you might want to do this. A prime example is if you are writing a class or defined resource type and want to give users the option to specifically request a parameter's default value. Some people have used `undef` to do this, but that's no good when dealing with parameters where `undef` would, itself, be a meaningful value. Others have used a value like the string `"UNSET"`, but this can be messy.

Using `default` in this scenario lets you distinguish among:

- A chosen "real" value.
- A chosen value of `undef` .
- Explicitly declining to choose a value, represented by `default` .

In other other words, `default` can be useful when you need a truly meaningless value.

#### The `Default` data type

The data type of `default` is `Default`. It matches only the value `default`, and takes no parameters.

Example:

| | |
|---|---|
| **`Variant[String, Default, Undef]`** | Matches `undef`, `default`, or any string. |

#### Related information

Data type syntax on page 326
Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Abstract data types on page 329
If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

### Resource and class references

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

The general form of a resource reference is:

- The resource type, capitalized. If the resource type includes a namespace separator `::`, then each segment must be capitalized.
- An opening square bracket `[`.
- The title of the resource as a string, or a comma-separated list of titles.
- A closing square bracket `]`.

For example, here is a reference to a file resource:

```
subscribe => File['/etc/ntp.conf'],
```

Here is a type with a multi-segment name:

```
before => Concat::Fragment['apache_port_header'],
```

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

### Class references

Class references work the same as resource references, but use the pseudo-resource type `Class` instead of some other resource type name:

```
require => Class['ntp::install'],
```

### Multi-resource references

Resource reference expressions with an array of titles or comma-separated list of titles refer to multiple resources of the same type. They evaluate to an array of single-title resource references. For example, here is a multi-resource reference:

```
require => File['/etc/apache2/httpd.conf', '/etc/apache2/magic', '/etc/
apache2/mime.types'],
```

And here is an equivalent multi-resource reference:

```
$my_files = ['/etc/apache2/httpd.conf', '/etc/apache2/magic', '/etc/apache2/
mime.types']
require => File[$my_files]
```

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

### Accessing attribute values

You can use a resource reference to access the values of a resource's attributes. To access a value, use square brackets and the name of an attribute (as a string). This works much like accessing hash values.

```
file { "/etc/first.conf":
  ensure => file,
  mode   => "0644",
  owner  => "root",
}

file { "/etc/second.conf":
  ensure => file,
  mode   => File["/etc/first.conf"]["mode"],
  owner  => File["/etc/first.conf"]["owner"],
}
```

The resource whose values you're accessing must exist.

Like referencing variables, attribute access depends on evaluation order: Puppet must evaluate the resource you're accessing before you try to access it. If it hasn't been evaluated yet, Puppet raises an evaluation error.

You can only access attributes that are valid for that resource type. If you try to access a nonexistent attribute, Puppet raises an evaluation error.

Puppet can read the values of only those attributes that are explicitly set in the resource's declaration. It can't read the values of properties that would have to be read from the target system. It also can't read the values of attributes that default to some predictable value. For example, in the code above, you wouldn't be able to access the value of the `path` attribute, even though it defaults to the resource's title.

Like with hash access, the value of an attribute whose value was never set is `undef`.

### Resource references as data types

If you've read the Data type syntax page, or perused the lower sections of the other data type pages, you might have noticed that resource references use the same syntax as values that represent data types. Internally, they're implemented the same way, and each resource reference is actually a data type.

For most users, this doesn't matter at all. You should treat resource references as a special case with a coincidentally similar syntax, and it'll make your life generally easier. But if you're interested in the details, see Resource types.

To restrict values for a class or defined type parameter so that users must provide your code a resource reference, do one of the following.

- To allow a resource reference of any resource type, use a data type of:

  ```
  Type[Resource]
  ```
- To allow resource references *and* class references, use a data type of:

  ```
  Type[Catalogentry]
  ```
- To allow a resource reference of a specific resource type — in this example, `file` — use one of the following:

  ```
  Type[File]              # Capitalized resource type name
  Type[Resource["file"]]  # Resource data type, with type name in parameter
   as a string
  Type[Resource[File]]    # Resource data type, with capitalized resource
   type name
  ```

  Any of these three options allow any `File['<TITLE>']` resource reference, while rejecting ones like `Service[<TITLE>]`.

## Resource types

Resource types are a special family of data types that behave differently from other data types. They are subtypes of the fairly abstract `Resource` data type. Resource references are a useful subset of this data type family.

In the Puppet language, there are never any values whose data type is one of these resource types. That is, you can never create an expression where `$my_value =~ Resource` evaluates to `true`. For example, a resource declaration — an expression whose value you might expect would be a resource — executes a side effect and then produces a resource reference as its value. A resource reference is a data type in this family of data types, rather than a value that has one of these data types.

In almost all situations, if one of these resource type data types is involved, it makes more sense to treat it as a special language keyword than to treat it as part of a hierarchy of data types. It does have a place in that hierarchy; it's just complicated, and you don't need to know it to do things in the Puppet language.

For that reason, the information on this page is provided for the sake of technical completeness, but learning it isn't critical to your ability to use Puppet successfully.

### Basics

Puppet automatically creates new known data type values for every resource type it knows about, including custom resource types and defined types.

These one-off data types share the name of the resource type they correspond to, with the first letter of every namespace segment capitalized. For example, the `file` type creates a data type called `File`.

Additionally, there is a parent `Resource` data type. All of these one-off data types are more-specific subtypes of `Resource`.

### Usage of resource types without a title

A resource data type can be used in the following places:

- The resource type slot of a resource declaration.
- The resource type slot of a resource default statement.

For example:

```
# A resource declaration using a resource data type:
File { "/etc/ntp.conf":
  mode  => "0644",
  owner => "root",
  group => "root",
}

# Equivalent to the above:
Resource["file"] { "/etc/ntp.conf":
  mode  => "0644",
  owner => "root",
  group => "root",
}

# A resource default:
File {
  mode  => "0644",
  owner => "root",
  group => "root",
}
```

### Usage of resource types with a title

If a resource data type includes a title, it acts as a resource reference, which are useful in several places. For more information, see Resource and class references on page 320.

### The <SOME ARBITRARY RESOURCE TYPE> data type

For each resource type `mytype` known to Puppet, there is a data type, `Mytype`. It matches no values that can be produced in the Puppet language. You can use parameters to restrict which values `Mytype` matches, but it still matches no values.

### Parameters

The full signature for a resource-type-corresponding data type `Mytype` is:

```
Mytype[<RESOURCE TITLE>]
```

This parameter is optional.

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Resource title | `String` | nothing | The title of some specific resource of this type. If provided, this turns this data type into a usable resource reference. |

Examples:

| | |
|---|---|
| **`File`** | The data type corresponding to the `file` resource type. |
| **`File['/tmp/filename.ext']`** | A resource reference to the `file` resource whose title is `/tmp/filename.ext`. |
| **`Type[File]`** | The data type that matches any resource references to `file` resources. This is useful for, for example, restricting the values of class or defined type parameters. |

### The `Resource` data type

There is also a general `Resource` data type, which all `<SOME ARBITRARY RESOURCE TYPE>` data types are more-specific subtypes of. Like the `Mytype`-style data types, it matches no values that can be produced in the Puppet language. You can use parameters to restrict which values `Resource` matches, but it still matches no values.

This is useful in the following uncommon circumstances:

- You need to interact with a resource type before you know its name. For example, you can do some clever business with the iteration functions to re-implement the `create_resources` function in the Puppet language, where your lambda receives arguments telling it to create resources of some resource type at runtime.
- Someone has somehow created a resource type whose name is invalid in the Puppet language, possibly by conflicting with a reserved word — you can use a `Resource` value to refer to that resource type in resource declarations and resource default statements, and to create resource references.

#### Parameters

The full signature for `Resource` is:

```
Resource[<RESOURCE TYPE>, <RESOURCE TITLE>...]
```

All of these parameters are optional. They must be listed in order; if you need to specify a later parameter, you must specify values for any prior ones.

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Resource type | `String` or `Resource` | nothing | A resource type, either as a string or a `Resource` data type value. If provided, this turns this data type into a resource-specific data type. `Resource[Mytype]` and `Resource["mytype"]` are identical to the data type `Mytype`. |

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 2 and higher | Resource title | `String` | nothing | The title of some specific resource of this type. If provided, this turns this data type into a usable resource reference or array of resource references. `Resource[Mytype, "mytitle"]` and `Resource["my "mytitle"]` are identical to the data type `Mytype["mytitle"]`. |

Examples:

| | |
|---|---|
| **`Resource[File]`** | The data type corresponding to the `file` resource type. |
| **`Resource[File, '/tmp/filename.ext']`** | A resource reference to the `file` resource whose title is `/tmp/filename.ext`. |
| **`Resource["file", '/tmp/filename.ext']`** | A resource reference to the `file` resource whose title is `/tmp/filename.ext`. |
| **`Resource[File, '/tmp/filename.ext', '/tmp/bar']`** | Equivalent to `[ File['/tmp/filename.ext'], File['/tmp/bar'] ]`. |
| **`Type[Resource[File]]`** | A synonym for the data type that matches any resource references to `file` resources. This is useful for, for example, restricting the values of class or defined type parameters. |
| **`Type[Resource["file"]]`** | Another synonym for the data type that matches any resource references to `file` resources. This is useful for, for example, restricting the values of class or defined type parameters. |

### The `Class` data type

The `Class` data type is roughly equivalent to the set of `Mytype` data types, except it is for classes. Like the `Mytype`-style data types, it matches no values that can be produced in the Puppet language. You can use parameters to restrict which values `Class` matches, but it will matches no values.

#### Parameters

The full signature for Class is:

```
Class[<CLASS NAME>]
```

This parameter is optional.

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Class name | `String` | nothing | The name of a class. If provided, this turns this data type into a usable class reference. |

Examples:

| | |
|---|---|
| `Class["apache"]` | A class reference to class `apache`. |
| **Type[Class]** | The data type that matches any class references. This is useful for, for example, restricting the values of class or defined type parameters. |

#### Related data types

The abstract `Catalogentry` data type is the supertype of `Resource` and `Class`. You can use `Type[Catalogentry]` as the data type for a class or defined type parameter that can accept both class references and resource references.

#### Related information

Resource and class references on page 320
Resource references identify a specific Puppet resource by its type and title. Several attributes, such as the relationship metaparameters, require resource references.

Data type syntax on page 326
Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Abstract data types on page 329
If you’re using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

## Data type syntax

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

You can use these special values to examine a piece of data or enforce rules. For example, you can test whether something is a string with the expression `$possible_string =~ String`, or specify that a class parameter requires string values with `class myclass (String $string_parameter = "default value") { ... }`.

#### Syntax

Data types are written as unquoted upper-case words, like `String`.

Data types sometimes take parameters, which make them more specific. For example, `String[8]` is the data type of strings with a minimum of eight characters.

Each known data type defines how many parameters it accepts, what values those parameters take, and the order in which they must be given. Some of the abstract types *require* parameters, and most types have some optional parameters available.

The general form of a data type is:

- An upper-case word matching one of the known data types.
- Sometimes, a set of parameters, which consists of:
  - An opening square bracket `[` after the type’s name. There can’t be any space between the name and the bracket.
  - A comma-separated list of values or expressions. Arbitrary whitespace is allowed, but you can’t have a trailing comma after the final value.
  - A closing square bracket `]`.

The following example uses an abstract data type `Variant`, which takes any number of data types as parameters. One of the parameters provided in the example is another abstract data type `Enum`, which takes any number of strings as parameters:

```
Variant[Boolean, Enum['true', 'false', 'running', 'stopped']]
```

**Note:** When parameters are required, you must specify them. The only situation when you can leave out required parameters is if you're referring to the type itself. For example, `Type[Variant]` is legal, even though `Variant` has required parameters.

### Usage

Data types are useful in parameter lists, match (`=~`) expressions, case statements, and selector expressions. There are also a few less common uses for them.

Specify data types in your Puppet code whenever you can, aligning them in columns. Class parameters should be typed wherever possible, and the type used should be as specific as possible. For example, use an `Enum` for input validation, instead of using a `String` and checking the contents of the string in the code. You have the option to specify `String[1]` instead of `String`, because you might want to enforce non-empty strings. You should also specify data types as deeply as possible, without affecting readability. If readability becomes a problem, consider creating a custom data type alias.

#### Parameter lists

Classes, defined types, and lambdas all let you specify parameters, which let your code request data from a user or some other source. Generally, your code expects each parameter to be a specific kind of data. You can enforce that expectation by putting a data type before that parameter's name in the parameter list. At evaluation time, Puppet raises an error if a parameter receives an illegal value.

For example, consider the following class. If you tried to set `$autoupdate` to a string like `"true"`, Puppet would raise an error, because it expects a Boolean value:

```
class ntp (
  Boolean $service_manage = true,
  Boolean $autoupdate     = false,
  String  $package_ensure = 'present',
  # ...
) {
  # ...
}
```

Abstract data types let you write more sophisticated and flexible restrictions. For example, this `$puppetdb_service_statusparameter` accepts values of `true`, `false`, `"true"`, `"false"`, `"running"`, and `"stopped"`, and raises an error for any other value:

```
class puppetdb::server (
  Variant[Boolean, Enum['true', 'false', 'running', 'stopped']]
    $puppetdb_service_status = $puppetdb::params::puppetdb_service_status,
) inherits puppetdb::params {
  # ...
}
```

#### Cases

Case statements and selector expressions allow data types as their cases. Puppet chooses a data type case if the control expression resolves to a value of that data type. For example:

```
$enable_real = $enable ? {
  Boolean => $enable,
  String  => str2bool($enable),
```

```
  Numeric => num2bool($enable),
  default => fail('Illegal value for $enable parameter'),
}
```

### Match expressions

The match operators `=~` and `!~` accept a data type on the right operand, and test whether the left operand is a value of that data type.

For example, `5 =~ Integer` and `5 =~ Integer[1,10]` both resolve to `true`.

### Less common uses

The built-in function `assert_type` takes a value and a data type, and raises errors if your code encounters an illegal value. Think of it as shorthand for an `if` statement with a non-match (`!~`) expression and a `fail()` function call.

You can also provide data types as both operands for the comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=`, to test whether two data types are equal, whether one is a subset of another, and so on.

### Obtaining data types

The built-in function `type` returns the type of any value. For example, `type(3)` returns `Integer[3,3]`.

### List of Puppet data types

The following data types are available in the Puppet language.

### Core data types

These are the "real" data types, which make up the most common values you'll interact with in the Puppet language.

| | | | |
|---|---|---|---|
| `String` | `Integer` | `Float` | `Numeric` |
| `Boolean` | `Array` | `Hash` | `Regexp` |
| `Sensitive` | `Undef` | `Default` | |

### Resource and class references

Resource references and class references are implemented as data types, although they behave somewhat differently from other values.

| |
|---|
| `Resource` and `Class` |

### Abstract data types

Abstract data types let you do more sophisticated or permissive type checking.

| | | | |
|---|---|---|---|
| `Scalar` | `Collection` | `Variant` | `Data` |
| `Pattern` | `Enum` | `Tuple` | `Struct` |
| `Optional` | `Catalogentry` | `Type` | `Any` |
| `Callable` | | | |

#### The Type data type

The data type of literal data type values is `Type`. By default, `Type` matches any value that represents a data type, such as `Integer`, `Integer[0,800]`, `String`, or `Enum["running", "stopped"]`. You can use parameters to restrict which values `Type` matches.

#### Parameters

The full signature for `Type` is:

```
Type[<ANY DATA TYPE>]
```

This parameter is optional.

| Position | Parameter | Data type | Default | Description |
|---|---|---|---|---|
| 1 | Any data type | `Type` | `Any` | A data type, which causes the resulting `Type` object to only match against that type or types that are more specific subtypes of that type. |

Examples:

**`Type`**
Matches any data type, such as `Integer`, `String`, `Any`, or `Type`.

**`Type[String]`**
Matches the data type `String`, as well as any of its more specific subtypes like `String[3]` or `Enum["running", "stopped"]`.

**`Type[Resource]`**
Matches any `Resource` data type — that is, any resource reference.

### Abstract data types

If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Each of Puppet's core data types has a corresponding value that represents that data type, which can be used to match values of that type in several contexts. Each of those core data types only match a particular set of values. They let you further restrict the values they'll match, but only in limited ways, and there's no way to *expand* the set of values they'll match. If you need to do this, use the corresponding abstract data type.

#### Related information

Data type syntax on page 326
Each value in the Puppet language has a data type, like "string." There is also a set of values *whose data type is "data type."* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

Values and data types on page 294

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

## Flexible data types

These abstract data types can match values with a variety of concrete data types. Some of them are similar to a concrete type but offer alternate ways to restrict them (for example, Enum), and some of them let you combine types and match a union of what they would individually match (for example, Variant and Optional).

### The Optional data type

The Optional data type wraps *one* other data type, and results in a data type that matches anything that type would match *plus* undef. This is useful for matching values that are allowed to be absent. It takes one required parameter.

The full signature for Optional is:

```
Optional[<DATA TYPE>]
```

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Data type | Type or String | none (you must specify a value) | The data type to add undef to. |

If you specify a string "my string" as the parameter, it's equivalent to using Optional[Enum["my string"]] — it matches only that exact string value or undef.

Optional[<DATA TYPE>] is equivalent to Variant[ <DATA TYPE>, Undef ].

Examples:

| | |
|---|---|
| **Optional[String]** | Matches any string or undef. |
| **Optional[Array[Integer[0, 10]]]** | Matches an array of integers between 0 and 10, or undef. |
| **Optional["present"]** | Matches the exact string "present" or undef. |

### The NotUndef data type

The NotUndef type matches any value except undef. It can also wrap one other data type, resulting in a type that matches anything the original type would match except undef. It accepts one optional parameter.

The full signature for NotUndef is:

```
NotUndef[<DATA TYPE>]
```

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Data type | Type or String | Any | The data type to subtract undef from. |

If you specify a string as a parameter for NotUndef, it's equivalent to writing NotUndef[Enum["my string"]] — it matches only that exact string value. This doesn’t actually subtract anything, because the Enum wouldn’t have matched undef anyway, but it's a convenient notation for mandatory keys in Struct schema hashes.

### The Variant data type

The Variant data type combines any number of other data types, and results in a type that matches the union of what any of those data types would match. It takes any number of parameters, and requires at least one.

The full signature for `Variant` is:

```
Variant[ <DATA TYPE>, (<DATA TYPE, ...) ]
```

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 and up | Data type | Type | none (required) | A data type to add to the resulting compound data type. You must provide at least one data type parameter, and can provide any number of additional ones. |

Examples:

| | |
|---|---|
| **`Variant[Integer, Float]`** | Matches any integer or floating point number (equivalent to `Numeric`). |
| **`Variant[Enum['true', 'false'], Boolean]`** | matches `'true'`, `'false'`, `true`, or `false`. |

### The `Pattern` data type

The Pattern data type only matches strings, but it provides an alternate way to restrict which strings it matches. It takes any number of regular expressions, and results in a data type that matches any strings that would match any of those regular expressions. It takes any number of parameters, and requires at least one.

The full signature for Pattern is:

```
Pattern[ <REGULAR EXPRESSION>, (<REGULAR EXPRESSION>, ...) ]
```

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 and up | Regular expression | `Regexp` | none (required) | A regular expression describing a set of strings that the resulting data type should match. You must provide at least one regular expression parameter, and can provide any number of additional ones. |

You can use capture groups in the regular expressions, but they won't cause any variables, like `$1`, to be set.

Examples:

| | |
|---|---|
| **`Pattern[/\A[a-z].*/]`** | Matches any string that begins with a lowercase letter. |
| **`Pattern[/\A[a-z].*/, /\ANone\Z/]`** | Matches the above or the exact string `None`. |

### The `Enum` data type

The `Enum` data type only matches strings, but it provides an alternate way to restrict which strings it matches. It takes any number of strings, and results in a data type that matches any string values that exactly match one of those strings. Unlike the `==` operator, this matching is case-sensitive. It takes any number of parameters, and requires at least one.

The full signature for `Enum` is:

```
Enum[ <OPTION>, (<OPTION>, ...) ]
```

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 and up | Option | `String` | none (required) | One of the literal string values that the resulting data type should match. You must provide at least one option parameter, and can provide any number of additional ones. |

Examples:

| | |
|---|---|
| **`Enum['stopped', 'running']`** | Matches the strings `'stopped'` and `'running'`, and no other values. |
| **`Enum['true', 'false']`** | Matches the strings `'true'` and `'false'`, and no other values. Does not match the boolean values `true` or `false` (without quotes). |

### The `Tuple` data type

The `Tuple` type only matches arrays, but it lets you specify different data types for every element of the array, in order. It takes any number of parameters, and requires at least one.

The full signature for `Tuple` is:

```
Tuple[ <CONTENT TYPE>, (<CONTENT TYPE>, ..., <MIN SIZE>, <MAX SIZE>) ]
```

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 and up | Content type | `Type` | none (required) | What kind of values the array contains at the given position. You must provide at least one content type parameter, and can provide any number of additional ones. |

| Position | Parameter | Data type | Default value | Description |
| --- | --- | --- | --- | --- |
| -2 (second-last) | Minimum size | `Integer` | number of content types | The minimum number of elements in the array. If this is smaller than the number of content types you provided, any elements beyond the minimum are optional; however, if present, they must still match the provided content types. This parameter accepts the value `default`, but this won't use the default value; instead, it means 0 (all elements optional). |
| -1 (last) | Maximum size | `Integer` | number of content types | The maximum number of elements in the array. You cannot specify a maximum without also specifying a minimum. If the maximum is larger than the number of content types you provided, it means the array can contain any number of additional elements, which all must match the last content type. This parameter accepts the value `default`, but this won't use the default value; instead, it means infinity (any number of elements matching the final content type).<br><br>Don't set the maximum smaller than the number of content types you provide. |

Examples:

| | |
|---|---|
| **`Tuple[String, Integer]`** | Matches a two-element array containing a string followed by an integer, like `["hi", 2]`. |
| **`Tuple[String, Integer, 1]`** | Matches the above **or** a one-element array containing only a string. |
| **`Tuple[String, Integer, 1, 4]`** | Matches an array containing one string followed by zero to three integers. |
| **`Tuple[String, Integer, 1, default]`** | Matches an array containing one string followed by any number of integers. |

### The `Struct` data type

The `Struct` type only matches hashes, but it lets you specify:

- The name of every allowed key.
- Whether each key is required or optional.
- The allowed data type for each of those keys' values.

It takes one mandatory parameter.

The full signature for `Struct` is:

```
Struct[<SCHEMA HASH>]
```

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 | Schema hash | `Hash[Variant[String, Optional, NotUndef], Type]` | none (required) | A hash that has all of the allowed keys and data types for the struct. |

A `Struct`'s schema hash must have the same keys as the hashes it matches. Each value must be a data type that matches the allowed values for that key.

The keys in a schema hash are usually strings. They can also be an `Optional` or `NotUndef` type with the key's name as their parameter.

If a key is a string, Puppet uses the *value*'s type to determine whether it's optional — because accessing a missing key resolves to the value `undef`, the key is optional if the value type accepts `undef` (like `Optional[Array]`).

Note that this doesn't distinguish between an explicit value of `undef` and an absent key. If you want to be more explicit, you can use `Optional['my_key']` to indicate that a key can be absent, and `NotUndef['my_key']` to make it mandatory. If you use one of these, a value type that accepts `undef` is only used to decide about explicit `undef` values, not missing keys.

The following example `Struct` matches hashes like `{mode => 'read', path => '/etc/fstab'}`. Both the `mode` and `path` keys are mandatory; `mode`'s value must be one of `'read'`, `'write'`, or `'update'`, and `path` must be a string of at least one character:

```
Struct[{mode => Enum[read, write, update],
        path => String[1]}]
```

The following data type would match the same values as the previous example, but the `path` key is optional. If present, `path` must match `String[1]` or `Undef`:

```
Struct[{mode => Enum[read, write, update],
        path => Optional[String[1]]}]
```

In the following data type, the `owner` key can be absent, but if it's present, it must be a string; a value of `undef` isn't allowed:

```
Struct[{mode            => Enum[read, write, update],
        path            => Optional[String[1]],
        Optional[owner] => String[1]}]
```

In the following data type, the owner key is mandatory, but it allows an explicit `undef` value:

```
Struct[{mode            => Enum[read, write, update],
        path            => Optional[String[1]],
        NotUndef[owner] => Optional[String[1]]}]
```

### Parent types

These abstract data types are the parents of multiple other types, and match values that would match *any* of their sub-types. They're useful when you have very loose restrictions but not *no* restrictions.

#### The `Scalar` data type

The `Scalar` data type matches *all* values of the following concrete data types:

- Numbers (both integers and floats)
- Strings
- Booleans
- Regular expressions

It doesn't match `undef`, `default`, resource references, arrays, or hashes. It takes no parameters.

`Scalar` is equivalent to `Variant[Integer, Float, String, Boolean, Regexp]`.

#### The `Data` data type

The `Data` data type matches any value that would match `Scalar`, but it also matches:

- `undef`
- Arrays that only conatin values that also match `Data`
- Hashes whose keys match `Scalar` and whose values also match `Data`

It doesn't match `default` or resource references. It takes no parameters.

`Data` is especially useful because it represents the subset of types that can be directly represented in almost all serialization format, such as JSON.

#### The `Collection` data type

The `Collection` type matches *any* array or hash, regardless of what kinds of values or keys it contains. It only partially overlaps with `Data` — there are values, such as an array of resource references, that match `Collection` but do not match `Data`.

`Collection` is equivalent to `Variant[Array[Any], Hash[Any, Any]]`.

#### The `Catalogentry` data type

The `Catalogentry` data type is the parent type of `Resource` and `Class`. Like those types, the Puppet language contains no values that it ever matches. However, the type `Type[Catalogentry]` matches any class reference or resource reference. It takes no parameters.

#### The Any data type

The `Any` data type matches *any* value of *any* data type.

#### Other types

These types aren't quite like the others.

#### The `Callable` data type

The `Callable` data type matches callable lambdas provided as function arguments.

There is no way to interact with `Callable` values in the Puppet language, but Ruby functions written to the function API (`Puppet::Functions`) can use `Callable` to inspect the lambda provided to the function.

The full signature for `Callable` is:

```
Callable[ (<DATA TYPE>, ...,) <MIN COUNT>, <MAX COUNT>, <BLOCK TYPE> ]
```

All of these parameters are optional.

| Position | Parameter | Data type | Default value | Description |
|---|---|---|---|---|
| 1 to n | Data type | `Type` | none | Any number of data types, representing the data type of each argument the lambda accepts. |
| -3 (third last) | Minimum count | `Integer` | 0 | The minimum number of arguments the lambda accepts. This parameter accepts the value `default`, which uses its default value 0. |
| -2 (second last) | Maximum count | `Integer` | infinity | The maximum number of arguments the lambda accepts. This parameter accepts the value `default`, which uses its default value, infinity. |
| -1 (last) | Block type | `Type[Callable]` | none | The `block_type` of the lambda. |

## Templates

Templates are documents that combine code, data, and literal text to produce a final rendered output. The goal of a template is to manage a complicated piece of text with simple inputs.

Templates are written in a specialized templating language that generates text from data. Use templates to manage the content of your Puppet configuration files via the `content` attribute of the `file` resource type.

Embedded Puppet (EPP) is a templating language based on the Puppet language. You can use EPP in Puppet 4 and higher, and with Puppet 3.5 through 3.8 with the future parser enabled. Puppet evaluates EPP templates with the `epp` and `inline_epp` functions.

Embedded Ruby (ERB) is a templating language based on Ruby. Puppet evaluates ERB templates with the `template` and `inline_template` functions.

## Using templates

Templates are written in a specialized templating language that generates text from data. Use templates to manage the content of your Puppet configuration files via the `content` attribute of the `file` resource type.

### Templating languages

Puppet supports two templating languages:

- Embedded Puppet (EPP) uses Puppet expressions in special tags. EPP works with Puppet 4.0 and later, and with Puppet 3.5 through 3.8 with future parser enabled.
- Embedded Ruby (ERB) uses Ruby code in tags, and requires some Ruby knowledge. ERB works with all Puppet versions.

### When to use a template

Templates are more powerful than normal strings, and less powerful than modeling individual settings as resources. Whether to use a template is mainly a question of the complexity of the work you're performing.

When you're managing simple config files, a template generally isn't necessary because strings in the Puppet language allow you to interpolate variables and expressions into text. For short and simple config files, you can often use a heredoc and interpolate a few variables, or do something like `${ $my_array.join(', ') }`.

Use a template if you're doing complex transformations (especially iterating over collections) or working with very large config files.

Some situations, however, are too complex for a template to be effective. For example, using several modules that each need to manage parts of the same config file is impractical with either templates or interpolated strings. For shared configuration like this, model each setting in the file as an individual resource, with either a custom resource type or an Augeas, concat, or `file_line` resource. This approach is similar to how core resource types like `ssh_authorized_key` and `mount` work.

## Creating templates using Embedded Puppet

Embedded Puppet (EPP) is a templating language based on the Puppet language. You can use EPP in Puppet 4 and higher, and with Puppet 3.5 through 3.8 with the future parser enabled. Puppet evaluates EPP templates with the `epp` and `inline_epp` functions.

### EPP structure and syntax

An EPP template looks like a plain-text document interspersed with tags containing Puppet expressions. When evaluated, these tagged expressions can modify text in the template. You can use Puppet variables in an EPP template to customize its output.

The following example shows parameter tags (<% |), non-printing expression tags (<%), expression-printing tags (<%=), and comment tags (<%#). A hyphen in a tag (`-`) strips leading or trailing whitespace when printing the evaluated template:

```
<%- | Boolean $keys_enable,
      String  $keys_file,
```

```
      Array   $keys_trusted,
      String  $keys_requestkey,
      String  $keys_controlkey
| -%>
<% if $keys_enable { -%>

<%# Printing the keys file, trusted key, request key, and control key: -%>
keys <%= $keys_file %>
<% unless $keys_trusted =~ Array[Data,0,0] { -%>
trustedkey <%= $keys_trusted.join(' ') %>
<% } -%>
<% if $keys_requestkey =~ String[1] { -%>
requestkey <%= $keys_requestkey %>
<% } -%>
<% if $keys_controlkey =~ String[1] { -%>
controlkey <%= $keys_controlkey %>
<% } -%>

<% } -%>
```

### EPP tags

Embedded Puppet (EPP) has two tags for Puppet code expressions, optional tags for parameters and comments, and a way to escape tag delimiters.

The following table provides an overview of the main tag types used with EPP. See the sections below for additional detail about each tag, including instructions on trimming whitespace and escaping special characters.

| I want to ... | EPP tag syntax |
|---|---|
| Insert the value of a single expression. | `<%= EXPRESSION %>` |
| Execute an expression without inserting a value. | `<% EXPRESSION %>` |
| Declare the template's parameters. | `<% \| PARAMETERS \| %>` |
| Add a comment. | `<%# COMMENT %>` |

Text outside a tag is treated as literal text, but is subject to any tagged Puppet code surrounding it. For example, text surrounded by a tagged `if` statement only appears in the output if the condition is true.

### Expression-printing tags

An expression-printing tag inserts the value of a single Puppet expression into the output.

| | |
|---|---|
| Opening tag | `<%=` |
| Closing tag | `%>` |
| Closing tag with trailing whitespace and line break trimming | `-%>` |

Example tag:

```
<%= $fqdn %>
```

An expression-printing tag must contain any single Puppet expression that resolves to a value, including plain variables, function calls, and arithmetic expressions. If the value isn't a string, Puppet automatically converts it to a string based on the rules for value interpolation in double-quoted strings.

All facts are available in EPP templates. For example, to insert the value of the `fqdn` and `hostname` facts in an EPP template for an Apache config file:

```
ServerName <%= $facts[fqdn] %>
ServerAlias <%= $facts[hostname] %>
```

### Non-printing tags

A non-printing tag executes the code it contains, but doesn't insert a value into the output.

| | |
|---|---|
| Opening tag | `<%` |
| Opening tag with indentation trimming | `<%-` |
| Closing tag | `%>` |
| Closing tag with trailing whitespace and line break trimming | `-%>` |

Non-printing tags that contain iterative and conditional expressions can affect the untagged text they surround.

For example, to insert text only if a certain variable was set, write:

```
<% if $broadcastclient == true { -%>
broadcastclient
<% } -%>
```

Expressions in non-printing tags don't have to resolve to a value or be a complete statement, but the tag must close at a place where it would be legal to write another expression. For example, this doesn't work:

```
<%# Syntax error: %>
<% $servers.each -%>
# some server
<% |$server| { %> server <%= server %>
<% } -%>
```

You must keep `|$server| {` inside the first tag, because you can't insert an arbitrary statement between a function call and its required block.

### Parameter tags

A parameter tag declares which parameters the template accepts. Each parameter can be typed and can have a default value.

<table>
<tr><td>Opening tag with indentation trimming</td><td><%- |</td></tr>
<tr><td>Closing tag with trailing whitespace and line break trimming</td><td>| -%></td></tr>
</table>

Example tag:

```
<%- | Boolean $keys_enable = false, String $keys_file = '' | -%>
```

The parameter tag is optional; if used, it must be the first content in a template. Always close the parameter tag with a right-trimmed delimiter (`-%>`) to avoid outputting a blank line. Literal text, line breaks, and non-comment tags cannot precede the template's parameter tag. (Comment tags that precede a parameter tag must use the right-trimming tag to trim trailing whitespace.)

The parameter tag's pair of pipe characters (`|`) contains a comma-separated list of parameters. Each parameter follows this format:

```
Boolean $keys_enable = false
```

- An optional data type, which restricts the allowed values for the parameter (defaults to `Any`)
- A variable name
- An optional equals (=) sign and default value, which must match the data type, if one was specified

Parameters with default values are optional, and can be omitted when the template is evaluated. If you want to use a default value of `undef`, make sure to also specify a data type that allows `undef`. For example, `Optional[String]` accepts `undef` as well as any string.

### Comment tags

A comment tag's contents do not appear in the template's output.

| | |
|---|---|
| Opening tag | <%# |
| Closing tag | %> |
| Closing tag with space trimming | -%> |

Example tag:

```
<%# This is a comment. %>
```

### Literal tag delimiters

If you need the template's final output to contain a literal <% or %>, you can escape the characters as <%% or %%>. The first literal tag is taken, and the rest of the line is treated as a literal. This means that `<%% Test %%>` in an EPP template would turn out as `<% Test %%>`, not `<% Test %>`.

### Accessing EPP variables

Embedded Puppet (EPP) templates can access variables with the `$variable` syntax used in Puppet.

A template works like a defined type:

- It has its own anonymous local scope.
- The parent scope is set to node scope (or top scope if there's no node definition).
- When you call the template (with the `epp` or `inline_epp` functions), you can use parameters to set variables in its local scope.
- Unlike Embedded Ruby (ERB) templates, EPP templates cannot directly access variables in the calling class without namespacing. Fully qualify variables or pass them in as parameters.

EPP templates can use short names to access global variables (like `$os` or `$trusted`) and their own local variables, but must use qualified names (like `$ntp::tinker`) to access variables from any class. The exception to this rule is `inline_epp`.

### Special scope rule for `inline_epp`

If you evaluate a template with the `inline_epp` function, and if the template has no parameters, either passed or declared, you can access variables from the calling class in the template by using the variables' short names. This exceptional behavior is only allowed if **all** of the above conditions are true.

### Should I use a parameter or a class variable?

Templates have two ways to use data:

- Directly access class variables, such as `$ntp::tinker`

- Use parameters passed at call time

Use class variables when a template is closely tied to the class that uses it, you don't expect it to be used anywhere else, and you need to use a **lot** of variables.

Use parameters when a template is used in several different places and you want to keep it flexible. Remember that declaring parameters with a tag makes a template's data requirements visible at a glance.

### EPP parameters

When you pass parameters when you call a template, the parameters become local variables inside the template. To use a parameter in this way, pass a hash as the last argument of the `epp` or `inline_epp` functions.

For example, calling this:

```
epp('example/example.epp', { 'logfile' => "/var/log/ntp.log" })
```

to evaluate this template:

```
<%- | Optional[String] $logfile = undef | -%>
<%# (Declare the $logfile parameter as optional) -%>

<% unless $logfile =~ Undef { -%>
logfile <%= $logfile %>
<% } -%>
```

The keys of the hash match the variable names you'll be using in the template, minus the leading `$` sign. Parameters must follow the normal rules for local variable names.

If the template uses a parameter tag, it **must** be the first content in a template and you can **only** pass the parameters it declares. Passing any additional parameters is a syntax error. However, if a template omits the parameter tag, you can pass it any parameters.

If a template's parameter tag includes any parameters without default values, they are mandatory. You must pass values for them when calling the template.

### Example EPP template

The following example is an EPP translation of the `ntp.conf.erb` template from the `puppetlabs-ntp` module.

```
# ntp.conf: Managed by puppet.
#
<% if $ntp::tinker == true and ($ntp::panic or $ntp::stepout) { -%>
# Enable next tinker options:
# panic - keep ntpd from panicking in the event of a large clock skew
# when a VM guest is suspended and resumed;
# stepout - allow ntpd change offset faster
tinker<% if $ntp::panic { %> panic <%= $ntp::panic %><% } %><% if
 $ntp::stepout { -%> stepout <%= $ntp::stepout %><% } %>
<% } -%>

<% if $ntp::disable_monitor == true { -%>
disable monitor
<% } -%>
<% if $ntp::disable_auth == true { -%>
disable auth
<% } -%>

<% if $ntp::restrict =~ Array[Data,1] { -%>
# Permit time synchronization with our time source, but do not
# permit the source to query or modify the service on this system.
<% $ntp::restrict.flatten.each |$restrict| { -%>
```

```
restrict <%= $restrict %>
<% } -%>
<% } -%>

<% if $ntp::interfaces =~ Array[Data,1] { -%>
# Ignore wildcard interface and only listen on the following specified
# interfaces
interface ignore wildcard
<% $ntp::interfaces.flatten.each |$interface| { -%>
interface listen <%= $interface %>
<% } -%>
<% } -%>

<% if $ntp::broadcastclient == true { -%>
broadcastclient
<% } -%>

# Set up servers for ntpd with next options:
# server - IP address or DNS name of upstream NTP server
# iburst - allow send sync packages faster if upstream unavailable
# prefer - select preferrable server
# minpoll - set minimal update frequency
# maxpoll - set maximal update frequency
<% [$ntp::servers].flatten.each |$server| { -%>
server <%= $server %><% if $ntp::iburst_enable == true { %> iburst<% } %><%
 if $server in $ntp::preferred_servers { %> prefer<% } %><% if $ntp::minpoll
 { %> minpoll <%= $ntp::minpoll %><% } %><% if $ntp::maxpoll { %> maxpoll <
%= $ntp::maxpoll %><% } %>
<% } -%>

<% if $ntp::udlc { -%>
# Undisciplined Local Clock. This is a fake driver intended for backup
# and when no outside source of synchronized time is available.
server   127.127.1.0
fudge    127.127.1.0 stratum <%= $ntp::udlc_stratum %>
restrict 127.127.1.0
<% } -%>

# Driftfile.
driftfile <%= $ntp::driftfile %>

<% unless $ntp::logfile =~ Undef { -%>
# Logfile
logfile <%= $ntp::logfile %>
<% } -%>

<% unless $ntp::peers =~ Array[Data,0,0] { -%>
# Peers
<% [$ntp::peers].flatten.each |$peer| { -%>
peer <%= $peer %>
<% } -%>
<% } -%>

<% if $ntp::keys_enable { -%>
keys <%= $ntp::keys_file %>
<% unless $ntp::keys_trusted =~ Array[Data,0,0] { -%>
trustedkey <%= $ntp::keys_trusted.join(' ') %>
<% } -%>
<% if $ntp::keys_requestkey =~ String[1] { -%>
requestkey <%= $ntp::keys_requestkey %>
<% } -%>
<% if $ntp::keys_controlkey =~ String[1] { -%>
controlkey <%= $ntp::keys_controlkey %>
<% } -%>
```

```
<% } -%>
<% [$ntp::fudge].flatten.each |$entry| { -%>
fudge <%= $entry %>
<% } -%>

<% unless $ntp::leapfile =~ Undef { -%>
# Leapfile
leapfile <%= $ntp::leapfile %>
<% } -%>
```

To call this template from a manifest (assuming that the template file is located in the `templates` directory of the `puppetlabs-ntp` module), add the following code to the manifest:

```
# epp(<FILE REFERENCE>, [<PARAMETER HASH>])
file { '/etc/ntp.conf':
  ensure  => file,
  content => epp('ntp/ntp.conf.epp'),
  # Loads /etc/puppetlabs/code/environments/production/modules/ntp/
templates/ntp.conf.epp
}
```

### Validating and previewing EPP templates

Before deploying a template, validate its syntax and render its output to make sure the template is producing the results you expect. Use the `puppet epp` compand-line tool for validating and rendering Embedded Puppet (EPP) templates.

#### EPP validation

To validate your template, run: `puppet epp validate <TEMPLATE NAME>`

The `puppet epp` command includes an action that checks EPP code for syntax problems. The `<TEMPLATE NAME>` can be a file reference or can refer to a `<MODULE NAME>/<TEMPLATE FILENAME>` as the `epp` function. If a file reference can also refer to a module, Puppet validates the module's template instead.

You can also pipe EPP code directly to the validator: `cat example.epp | puppet epp validate`

The command is silent on a successful validation. It reports and halts on the first error it encounters. For information on how to modify this default behavior, see the command's man page.

#### EPP rendering

To render your template, run: `puppet epp render <TEMPLATE NAME>`

You can render EPP from the command line with `puppet epp render`. If Puppet can evaluate the template, it outputs the result.

If your template relies on specific parameters or values to function, you can simulate those values by passing a hash to the `--values` option. For example:

```
puppet epp render example.epp --values '{x => 10, y => 20}'
```

You can also render inline EPP by using the `-e` flag or piping EPP code to `puppet epp render`, and even simulate facts using YAML. For details, see the command's man page.

### Evaluating EPP templates

After you have an EPP template, you can pass it to a function that evaluates it and returns a final string. The actual template can be either a separate file or a string value.

#### Evaluating EPP templates that are in a template file

Put template files in the `templates` directory of a module. EPP files use the `.epp` extension.

To use a EPP template file, evaluate it with the `epp` function. For example:

```
# epp(<FILE REFERENCE>, [<PARAMETER HASH>])
file { '/etc/ntp.conf':
  ensure  => file,
  content => epp('ntp/ntp.conf.epp', {'service_name' => 'xntpd',
 'iburst_enable' => true}),
  # Loads /etc/puppetlabs/code/environments/production/modules/ntp/
templates/ntp.conf.epp
}
```

The first argument to the function is the *file reference*: a string in the form '<MODULE>/<FILE>', which loads <FILE> from <MODULE>'s templates directory. For example, the file reference `ntp/ntp.conf.epp` loads the <MODULES DIRECTORY>/ntp/templates/ntp.conf.epp file.

Some EPP templates declare parameters, and you can provide values for them by passing a *parameter hash* to the `epp` function.

The keys of the hash must be valid local variable names (minus the `$`). Inside the template, Puppet creates variables with those names and assign their values from the hash. For example, with a parameter hash of `{'service_name' => 'xntpd', 'iburst_enable' => true}`, an EPP template would receive variables called `$service_name` and `$iburst_enable`.

When structuring your parameter hash, remember:

- If a template declares any mandatory parameters, you **must** set values for them with a parameter hash.
- If a template declares any optional parameters, you can choose to provide values or let them use their defaults.
- If a template declares **no** parameters, you can pass any number of parameters with any names; otherwise, you can only choose from the parameters requested by the template.

#### Evaluating EPP template strings

If you have a string value that contains template content, you can evaluate it with the `inline_epp` function.

In older versions of Puppet, inline templates were mostly used to get around limitations — tiny Ruby fragments were useful for transforming and manipulating data before Puppet had iteration functions like `map` or puppetlabs/stdlib functions like `chomp` and `keys`.

In modern versions of Puppet, inline templates are usable in some of the same situations template files are. Because the heredoc syntax makes it easy to write large and complicated strings in a manifest, you can use `inline_epp` to reduce the number of files needed for a simple module that manages a small config file.

For example:

```
$ntp_conf_template = @(END)
...template content goes here...
END

# inline_epp(<TEMPLATE STRING>, [<PARAMETER HASH>])
file { '/etc/ntp.conf':
  ensure  => file,
  content => inline_epp($ntp_conf_template, {'service_name' => 'xntpd',
 'iburst_enable' => true}),
}
```

Some EPP templates declare parameters, and you can provide values for them by passing a *parameter hash* to the `epp` function.

The keys of the hash must be valid local variable names (minus the `$`). Inside the template, Puppet creates variables with those names and assign their values from the hash. For example, with a parameter hash of `{'service_name' => 'xntpd', 'iburst_enable' => true}`, an EPP template would receive variables called `$service_name` and `$iburst_enable`.

When structuring your parameter hash, remember:

- If a template declares any mandatory parameters, you **must** set values for them with a parameter hash.
- If a template declares any optional parameters, you can choose to provide values or let them use their defaults.
- If a template declares **no** parameters, you can pass any number of parameters with any names; otherwise, you can only choose from the parameters requested by the template.

## Creating templates using Embedded Ruby

Embedded Ruby (ERB) is a templating language based on Ruby. Puppet evaluates ERB templates with the `template` and `inline_template` functions.

**Note:** If you've used ERB in other projects, it might have had different features enabled. This page describes how ERB works with Puppet.

### ERB structure and syntax

An ERB template looks like a plain-text document interspersed with tags containing Ruby code. When evaluated, this tagged code can modify text in the template.

Puppet passes data to templates via special objects and variables, which you can use in the tagged Ruby code to control the templates' output. The following example shows non-printing tags (<%), expression-printing tags (<%=), and comment tags (<%#). A hyphen in a tag (`-`) strips leading or trailing whitespace when printing the evaluated template:

```
<% if @keys_enable -%>
<%# Printing the keys file, trusted key, request key, and control key: -%>
keys <%= @keys_file %>
<% unless @keys_trusted.empty? -%>
trustedkey <%= @keys_trusted.join(' ') %>
<% end -%>
<% if @keys_requestkey != '' -%>
requestkey <%= @keys_requestkey %>
<% end -%>
<% if @keys_controlkey != '' -%>
controlkey <%= @keys_controlkey %>
<% end -%>

<% end -%>
```

### ERB tags

Embedded Ruby (ERB) has two tags for Ruby code expressions, a tag for comments, and a way to escape tag delimiters.

The following table provides an overview of the main tag types used with ERB. See the sections below for additional detail about each tag, including instructions on trimming whitespace and escaping special characters.

| I want to ... | EPP tag syntax |
|---|---|
| Insert the value of a single expression. | `<%= EXPRESSION %>` |
| Execute an expression without inserting a value. | `<% EXPRESSION %>` |
| Add a comment. | `<%# COMMENT %>` |

Text outside a tag is treated as literal text, but is subject to any tagged Ruby code surrounding it. For example, text surrounded by a tagged `if` statement only appears in the output if the condition is true.

### Expression-printing tags

An expression-printing tag inserts the value into the output.

| | |
|---|---|
| Opening tag | `<%=` |
| Closing tag | `%>` |
| Closing tag with trailing whitespace and line break trimming | `-%>` |

Example tag:

```
<%= @fqdn %>
```

It must contain a snippet of Ruby code that resolves to a value; if the value isn't a string, it is automatically converted to a string using its `to_s` method.

For example, to insert the value of the `fqdn` and `hostname` facts in an EPP template for an Apache config file:

```
ServerName <%= @fqdn %>
ServerAlias <%= @hostname %>
```

### Non-printing tags

A non-printing tag executes the code it contains, but doesn't insert a value into the output.

| | |
|---|---|
| Opening tag | `<%` |
| Opening tag with indentation trimming | `<%-` |
| Closing tag | `%>` |
| Closing tag with trailing whitespace and line break trimming | `-%>` |

Non-printing tags that contain iterative and conditional expressions can affect the untagged text they surround.

For example, to insert text only if a certain variable was set, write:

```
<% if @broadcastclient == true -%>
broadcastclient
<% end -%>
```

Expressions in non-printing tags don't have to resolve to a value or be a complete statement, but the tag must close at a place where it would be legal to write another expression. For example, this doesn't work:

```
<%# Syntax error: %>
<% @servers.each -%>
# some server
<% do |server| %>server <%= server %>
<% end -%>
```

You must keep `do |server|` inside the first tag, because you can't insert an arbitrary statement between a function call and its required block.

### Comment tags

A comment tag's contents do not appear in the template's output.

| | |
|---|---|
| Opening tag | `<%#` |
| Closing tag | `%>` |
| Closing tag with line break trimming | `-%>` |

Example tag:

```
<%# This is a comment. %>
```

### Literal tag delimiters

If you need the template's final output to contain a literal `<%` or `%>`, you can escape the characters as `<%%` or `%%>`. The first literal tag is taken, and the rest of the line is treated as a literal. This means that `<%% Test %%>` in an ERB template would turn out as `<% Test %%>`, not `<% Test %>`.

### Accessing Puppet variables

ERB templates can access Puppet variables. This is the main source of data for templates.

An ERB template has its own local scope, and its parent scope is set to the class or defined type that evaluates the template. This means a template can use short names for variables from that class or type, but it can't insert new variables into it.

There are two ways to access variables in an ERB template:

- `@variable`
- `scope['variable']` and its older equivalent, `scope.lookupvar('variable')`

#### `@variable`

All variables in the current scope (including global variables) are passed to templates as Ruby instance variables, which begin with "at" signs (`@`). If you can access a variable by its short name in the surrounding manifest, you can access it in the template by replacing its `$` sign with an `@`, so that `$os` becomes `@os`, and `$trusted` becomes `@trusted`.

This is the most legible way to access variables, but it doesn't support variables from other scopes. For that, you need to use the `scope` object.

#### `scope['variable']` or `scope.lookupvar('variable')`

Puppet also passes templates an object called `scope`, which can access all variables (including out-of-scope variables) with a hash-style access expression. For example, to access `$ntp::tinker` you would use `scope['ntp::tinker']`.

Another way to use the `scope` object is to call its `lookupvar` method and pass the variable's name as its argument, as in `scope.lookupvar('ntp::tinker')`. This is exactly equivalent to the above, if slightly less convenient. This usage predates the hash-style indexing added in Puppet 3.0.

### Puppet data types in Ruby

Puppet's data types are converted to Ruby classes as follows:

| Puppet type | Ruby class |
|---|---|
| Boolean | `Boolean` |
| String | `String` |

| Puppet type | Ruby class |
| --- | --- |
| Number | Subtype of `Numeric` |
| Array | `Array` |
| Hash | `Hash` |
| Default | `Symbol` (value `:default`) |
| Regexp | `Regexp` |
| Resource reference | `Puppet::Pops::Types::PResourceType` or `Puppet::Pops::Types::PHostClassType` |
| Lambda (code block) | `Puppet::Pops::Evaluator::Closure` |
| Data type (type) | A type class under `Puppet::Pops::Types`, such as `Puppet::Pops::Types::PIntegerType` |
| Undef | `NilClass` (value `nil`) |
| | **Note:** If a Puppet variable was never defined, its value is `undef`, which means its value in a template is `nil`. |

## Using Ruby in ERB templates

To manipulate and print data in ERB templates, you'll need to know some Ruby. A full introductory Ruby tutorial is outside the scope of these docs, but this page provides an overview of Ruby basics commonly used in ERB templates.

### Using `if` statements

The `if ... end` statement in Ruby lets you write conditional text. Put the control statements in non-printing tags, and the conditional text between the tags:

```
<% if <CONDITION> %> text goes here <% end %>
```

For example:

```
<% if @broadcast != "NONE" %>broadcast <%= @broadcast %><% end %>
```

The general format of an `if` statement is:

```
if <CONDITION>
  ... code ...
elsif <CONDITION>
  ... other code ...
end
```

### Using iteration

Ruby lets you iterate over arrays and hashes with the `each` method. This method takes a block of code and executes it one time for each element in the array or hash. In a template, untagged text is treated as part of the code that gets repeated. You can think of literal text as an instruction, telling the evaluator to insert that text into the final output.

To write a block of code in Ruby, use either `do |arguments| ... end` or `{|arguments| ... }`. Note that this is different from Puppet lambdas — but they work similarly.

```
<% @values.each do |val| -%>
Some stuff with <%= val %>
<% end -%>
```

If $values was set to ['one', 'two'], this example would produce:

```
Some stuff with one
Some stuff with two
```

This example also trims line breaks for the non-printing tags, so they won't appear as blank lines in the output.

### Manipulating data

Your templates generally use data from Puppet variables. These values almost always be strings, numbers, arrays, and hashes.

These become the equivalent Ruby objects when you access them from an ERB template.

For information about the ways you can transform these objects, see the Ruby documentation for strings, integers, arrays, and hashes.

Also, note that the special undef value in Puppet becomes the special nil value in Ruby in ERB templates.

### Calling Puppet functions from ERB templates

You can use Puppet functions inside ERB templates by calling the scope.call_function(<NAME>, <ARGS>) method.

This method takes two arguments:

- The name of the function, as a string.
- All arguments to the function, as an array. This must be an array even for one argument or zero arguments.

For example, to evaluate one template inside another:

```
<%= scope.call_function('template', ["my_module/template2.erb"]) %>
```

To log a warning using the Puppet logging system, so that the warning appears in reports:

```
<%= scope.call_function('warning', ["Template was missing some data; this
 config file might be malformed."]) %>
```

**Note:**

scope.call_function was added in Puppet 4.2.

Previous versions of Puppet created a function_<NAME> method on the scope object for each function. These could be called with an arguments array, such as <%= scope.function_template(["my_module/template2.erb"]) %>.

While this method still works in Puppet 4.2 and later versions, the auto-generated methods don't support the modern function APIs, which are now used by the majority of built-in functions.

### Example ERB template

The following example is taken from the puppetlabs-ntp module.

```
# ntp.conf: Managed by puppet.
#
<% if @tinker == true and (@panic or @stepout) -%>
# Enable next tinker options:
# panic - keep ntpd from panicking in the event of a large clock skew
# when a VM guest is suspended and resumed;
# stepout - allow ntpd change offset faster
tinker<% if @panic -%> panic <%= @panic %><% end %><% if @stepout -%>
 stepout <%= @stepout %><% end %>
<% end -%>

<% if @disable_monitor == true -%>
```

```
disable monitor
<% end -%>
<% if @disable_auth == true -%>
disable auth
<% end -%>

<% if @restrict != [] -%>
# Permit time synchronization with our time source, but do not
# permit the source to query or modify the service on this system.
<% @restrict.flatten.each do |restrict| -%>
restrict <%= restrict %>
<% end -%>
<% end -%>

<% if @interfaces != [] -%>
# Ignore wildcard interface and only listen on the following specified
# interfaces
interface ignore wildcard
<% @interfaces.flatten.each do |interface| -%>
interface listen <%= interface %>
<% end -%>
<% end -%>

<% if @broadcastclient == true -%>
broadcastclient
<% end -%>

# Set up servers for ntpd with next options:
# server - IP address or DNS name of upstream NTP server
# iburst - allow send sync packages faster if upstream unavailable
# prefer - select preferrable server
# minpoll - set minimal update frequency
# maxpoll - set maximal update frequency
<% [@servers].flatten.each do |server| -%>
server <%= server %><% if @iburst_enable == true -%> iburst<% end %><% if
 @preferred_servers.include?(server) -%> prefer<% end %><% if @minpoll -%>
 minpoll <%= @minpoll %><% end %><% if @maxpoll -%> maxpoll <%= @maxpoll %><
% end %>
<% end -%>

<% if @udlc -%>
# Undisciplined Local Clock. This is a fake driver intended for backup
# and when no outside source of synchronized time is available.
server   127.127.1.0
fudge    127.127.1.0 stratum <%= @udlc_stratum %>
restrict 127.127.1.0
<% end -%>

# Driftfile.
driftfile <%= @driftfile %>

<% unless @logfile.nil? -%>
# Logfile
logfile <%= @logfile %>
<% end -%>

<% unless @peers.empty? -%>
# Peers
<% [@peers].flatten.each do |peer| -%>
peer <%= peer %>
<% end -%>
<% end -%>

<% if @keys_enable -%>
```

```
keys <%= @keys_file %>
<% unless @keys_trusted.empty? -%>
trustedkey <%= @keys_trusted.join(' ') %>
<% end -%>
<% if @keys_requestkey != '' -%>
requestkey <%= @keys_requestkey %>
<% end -%>
<% if @keys_controlkey != '' -%>
controlkey <%= @keys_controlkey %>
<% end -%>

<% end -%>
<% [@fudge].flatten.each do |entry| -%>
fudge <%= entry %>
<% end -%>

<% unless @leapfile.nil? -%>
# Leapfile
leapfile <%= @leapfile %>
<% end -%>
```

### Validating ERB templates

Before deploying a template, validate its syntax and render its output to make sure the template is producing the results you expect. Use the Ruby `erb` command to check Embedded Ruby (ERB) syntax.

#### ERB validation

To validate your ERB template, pipe the output from the `erb` command into `ruby`:

```
erb -P -x -T '-' example.erb | ruby -c
```

The `-P` switch ignores lines that start with '%', the `-x` switch outputs the template's Ruby script, and `-T '-'` sets the trim mode to be consistent with Puppet's behavior. This output gets piped into Ruby's syntax checker (`-c`).

If you need to validate many templates quickly, you can implement this command as a shell function in your shell's login script, such as `.bashrc`, `.zshrc`, or `.profile`:

```
validate_erb() {
  erb -P -x -T '-' $1 | ruby -c
}
```

You can then run `validate_erb example.erb` to validate an ERB template.

### Evaluating ERB templates

After you have an ERB template, you can pass it to a function that evaluates it and returns a final string. The actual template can be either a separate file or a string value.

#### Evaluating ERB templates that are in a template file

Put template files in the `templates` directory of a module. ERB files use the `.erb` extension.

To use a ERB template file, evaluate it with the `template` function. For example:

```
# template(<FILE REFERENCE>, [<ADDITIONAL FILES>, ...])
file { '/etc/ntp.conf':
  ensure  => file,
  content => template('ntp/ntp.conf.erb'),
  # Loads /etc/puppetlabs/code/environments/production/modules/ntp/
templates/ntp.conf.erb
}
```

The first argument to the function is the *file reference*: a string in the form `'<MODULE>/<FILE>'`, which loads `<FILE>` from `<MODULE>`'s templates directory. For example, the file reference `activemq/amq/activemq.xml.erb` loads the `<MODULES DIRECTORY>/activemq/templates/amq/activemq.xml.erb` file.

The template function can take any number of additional template files, and concatenate their outputs together to produce the final string.

### Evaluating ERB template strings

If you have a string value that contains template content, you can evaluate it with the `inline_template` function.

In older versions of Puppet, inline templates were mostly used to get around limitations — tiny Ruby fragments were useful for transforming and manipulating data before Puppet had iteration functions like `map` or puppetlabs/stdlib functions like `chomp` and `keys`.

In modern versions of Puppet, inline templates are usable in some of the same situations template files are. Because the heredoc syntax makes it easy to write large and complicated strings in a manifest, you can use `inline_epp` to reduce the number of files needed for a simple module that manages a small config file.

For example:

```
$ntp_conf_template = @(END)
...template content goes here...
END

# inline_template(<TEMPLATE STRING>, [<ADDITIONAL STRINGS>, ...])
file { '/etc/ntp.conf':
  ensure  => file,
  content => inline_template($ntp_conf_template),
}
```

The `inline_template` function can take any number of additional template strings, and concatenate their outputs together to produce the final value.

## Advanced constructs

Advanced Puppet language constructs help you write simpler and more effective Puppet code by reducing complexity.

- Iteration and loops on page 353

Looping and iteration features help you write more succinct code, and use data more effectively.

- Lambdas on page 355

Lambdas are blocks of Puppet code passed to functions. When a function receives a lambda, it provides values for the lambda's parameters and evaluates its code. If you use other programming languages, think of lambdas as anonymous functions that are passed to other functions.

- Resource default statements on page 358

Resource default statements enable you to set default attribute values for a given resource type. Resource declarations within the area of effect that omits those attributes inherit the default values.

- Resource collectors on page 358

Resource collectors select a group of resources by searching the attributes of each resource in the catalog, even resources which haven't yet been declared at the time the collector is written. Collectors realize virtual resources, are used in chaining statements, and override resource attributes. Collectors have an irregular syntax that enables them to function as a statement and a value.

- Virtual resources on page 360

A virtual resource declaration specifies a desired state for a resource without enforcing that state. Puppet manages the resource by realizing it elsewhere in your manifests. This divides the work done by a normal resource declaration into two steps. Although virtual resources are declared one time, they can be realized any number of times, similar to a class.

- Exported resources on page 362
An exported resource declaration specifies a desired state for a resource, and publishes the resource for use by other nodes. It does not manage the resource on the target system. Any node, including the node that exports it, can collect the exported resource and manage its own copy of it.
- Tags on page 364
Tags are useful for collecting resources, analyzing reports, and restricting catalog runs. Resources, classes, and defined type instances can have multiple tags associated with them, and they receive some tags automatically.
- Run stages on page 366
Run stages are an additional way to order resources. Groups of classes run before or after everything else, without having to explicitly create relationships with other classes. The run stage feature has two parts: a `stage` resource type, and a `stage` metaparameter, which assigns a class to a named run stage.

## Iteration and loops

Looping and iteration features help you write more succinct code, and use data more effectively.

### Iteration functions

Iteration features are implemented as functions that accept blocks of code (lambdas). When a lambda requires extra information, pass it to a function that provides the information and to evaluate the code, possibly multiple times. This differs from some other languages where looping constructs are special keywords. In the Puppet code, they're functions.

**Tip:** Iteration functions take an array or a hash as their main argument, and iterate over its values.

These functions accept a block of code and run it in a specific way:

- `each` - Repeats a block of code a number of times, using a collection of values to provide different parameters each time.
- `slice` - Repeats a block of code a number of times, using groups of values from a collection as parameters.
- `filter` - Uses a block of code to transform a data structure by removing non-matching elements.
- `map` - Uses a block of code to transform every value in a data structure.
- `reduce` - Uses a block of code to create a new value, or data structure, by combining values from a provided data structure.
- `with` - Evaluates a block of code one time, isolating it in its own local scope. It doesn't iterate, but has a family resemblance to the iteration functions.

The `each`, `filter`, and `map` functions accept a lambda with either one or two parameters. Depending on the number of parameters, and the type of data structure you're iterating over, the values passed into a lambda vary:

| Collection type | Single parameter | Two parameters |
|---|---|---|
| Array | `<VALUE>` | `<INDEX>, <VALUE>` |
| Hash | `[<KEY>, <VALUE>]` (two-element array) | `<KEY>, <VALUE>` |

For example:

```
['a','b','c'].each |Integer $index, String $value| { notice("${index} =
 ${value}") }
```

Results in:

```
Notice: Scope(Class[main]): 0 = a
Notice: Scope(Class[main]): 1 = b
Notice: Scope(Class[main]): 2 = c
```

See the `slice` and `reduce` documentation for information on how these functions handle parameters differently.

Hashes preserve the order in which their keys and values were written. When iterating over a hash's members, the loops occur in the order that they are written. When interpolating a hash into a string, the resulting string is also constructed in the same order.

For information about the syntax of function calls, see the functions documentation, or the lambdas documentation for information about the syntax of code blocks that you pass to functions.

### Declaring resources

The focus of the Puppet language is declaring resources, so most people want to use iteration to declare many similar resources at the same time. In this example, there is an array of command names to be used in each symlink's path and target. The `each` function makes this succinct.

```
$binaries = ['facter', 'hiera', 'mco', 'puppet', 'puppetserver']

$binaries.each |String $binary| {
  file {"/usr/bin/${binary}":
    ensure => link,
    target => "/opt/puppetlabs/bin/${binary}",
  }
}
```

### Iteration with defined resource types

In previous versions of Puppet, iteration functions did not exist and lambdas weren't supported. By writing defined resource types and using arrays as resource titles you could achieve a clunkier form of iteration.

Similar to the declaring resources example, include an unique defined resource type in the `symlink.pp` file:

```
define puppet::binary::symlink ($binary = $title) {
  file {"/usr/bin/${binary}":
    ensure => link,
    target => "/opt/puppetlabs/bin/${binary}",
  }
}
```

Use the defined type for the iteration somewhere ele in your manifest file:

```
$binaries = ['facter', 'hiera', 'mco', 'puppet', 'puppetserver']

puppet::binary::symlink { $binaries: }
```

The main problems with this approach are:

- The block of code doing the work was separated from the place where you used it, which makes a simple task complicated.
- Every type of thing to iterate over would require its own one-off defined type.

The current Puppet style of iteration is much improved, but you might encounter code that uses this old style, and might have to use it to target older versions of Puppet.

### Using iteration to transform data

To transform data into more useful forms, use iteration. For example:

This returns `[1,3]`:

```
$filtered_array = [1,20,3].filter |$value| { $value < 10 }
```

This returns `6`:

```
$sum = reduce([1,2,3]) |$result, $value|  { $result + $value }
```

This returns `{"key1"=>"first value", "key2"=>"second value", "key3"=>"third value"}`:

```
$hash_as_array = ['key1', 'first value',
                  'key2', 'second value',
                  'key3', 'third value']

$real_hash = $hash_as_array.slice(2).reduce( {} ) |Hash $memo, Array $pair|
 {
  $memo + $pair
}
```

## Lambdas

Lambdas are blocks of Puppet code passed to functions. When a function receives a lambda, it provides values for the lambda's parameters and evaluates its code. If you use other programming languages, think of lambdas as anonymous functions that are passed to other functions.

### Location

Lambdas are used only in function calls. They cannot be assigned to variables, and are not valid anywhere else in the Puppet language. While any function accepts a lambda, only some functions do anything with them. For information on useful lambda-accepting functions, see Iteration and loops.

### Syntax

Lambdas consist of a list of parameters surrounded by pipe (`|`) characters, followed by a block of arbitrary Puppet code in curly braces. They must be used as part of a function call.

```
$binaries = ['facter', 'hiera', 'mco', 'puppet', 'puppetserver']

# function call with lambda:
$binaries.each |String $binary| {
  file {"/usr/bin/${binary}":
    ensure => link,
    target => "/opt/puppetlabs/bin/${binary}",
  }
}
```

The general form of a lambda is:

- A mandatory parameter list, which can be empty. This consists of:
  - An opening pipe character (`|`).
  - A comma-separated list of zero or more parameters (for example, `String $myparam = "default value"`). Each parameter consists of:
    - An optional data type, which restricts the values it allows (defaults to `Any`).
    - A variable name to represent the parameter, including the `$` prefix.
    - An optional equals (`=`) sign and default value.
    - An opening pipe character (`|`).
  - Optionally, another comma and an extra arguments parameter (for example, `String *$others = ["default one", "default two"]`), which consists of:
    - An optional data type, which restricts the values allowed for extra arguments (defaults to `Any`).
    - An asterisk character (`*`).
    - A variable name to represent the parameter, including the `$` prefix.
    - An optional equals (`=`) sign and default value, which can be one value that matches the specified data type, or an array of values that all match the data type.
  - An optional trailing comma after the last parameter.
  - A closing pipe character (`|`).
- An opening curly brace.
- A block of arbitrary Puppet code.
- A closing curly brace.

### Parameters and variables

When functions call the lambda it sets values for the list of parameters that a lambda contains. and each parameter can be used as a variable.

Functions pass lambda parameters by position, similar to passing arguments in a function call. Each function decides how many parameters, and in what order, it passes to a lambda. See the function's documentation for details.

**Important:** The order of parameters is important and there are no restrictions on naming — unlike class or defined type parameters, where the names are the main interface for users.

Within the parameter list, the data type preceding a parameter is optional. To ensure the correct data is included, Puppet checks the parameter value at runtime, and raises an error when the value is illegal. When no data type is provided, values of any data type are accepted by the parameter.

When a parameter contains a default value, it's optional — the lambda uses the default value when the caller doesn't provide a value for that parameter.

**Important:** Parameters are passed by position. Optional parameters must be poistioned after the required parameters, otherwise it causes an evaluation error. When you have multiple optional parameters, the later ones only receive values if all of the prior ones do.

The final parameter of a lambda can be a special extra arguments parameter, which collects an unlimited number of extra arguments into an array. This is useful when you don't know in advance how many arguments the caller provides.

To specify that the last parameter collects extra arguments, write an asterisk (`*`) in front of its name in the parameter list (like `*$others`). An extra arguments parameter is always optional. You can't put an asterisk (`*`) in front of any parameter except the last one. The value of an extra arguments parameter is always an array, containing every argument in excess of the earlier parameters. If there are no extra arguments and no default value, it will be an empty array.

An extra arguments parameter can contain a default value, which has automatic array wrapping for convenience:

- When the provided default is a non-array value, the real default is a single-element array containing that value.
- When the provided default is an array, the real default is that array.

An extra arguments parameter can also contain a data type. Puppet uses this data type to validate the elements of the array. When you specify a data type of `String`, the final data type of the extra arguments parameter will be `Array[String]`.

### Behavior

Similar to a defined type, a lambda delays evaluation of the Puppet code it contains and makes it available for later. Unlike defined types, lambdas are not directly invoked by a user. The user provides a lambda to some other piece of code (a function), and that code decides:

- Whether (and when) to call/evaluate the lambda.
- How many times to call it.
- What values its parameters should have.
- What to do with any values it produces.

Some functions call a single lambda multiple times and provide different parameter values each time. For information on how a particular function uses its lambda, see its documentation. In this version of the Puppet language, calling a lambda is to pass it to a function that calls it.

You must use unique resource declarations in the body of a lambda, duplicate resources cause compilation failures. This means that when a function calls its lambda multiple times, any resource titles in the lambda should include a parameter value that changes with every call.

In this example, we use the `$binary` parameter in the title of the lambda's `file` resource:

```
file {"/usr/bin/$binary":
  ensure => link,
  target => "/opt/puppetlabs/bin/$binary",
}
```

When the `each` function is called, the array we pass has no repeated values to ensure unique `file` resources. However, if we are working with an array that came from less reliable external data, we could use the `unique` function from `stdlib` to protect against duplicates. This uniqueness requirement is similar to defined types, which are also blocks of Puppet code that are evaluated multiple times.

Each time a lambda is called it produces the value of the last expression in the code block. The function that calls the lambda has access to this value, but not every function does anything with it. Some functions return it, some transform it, some ignore it, and some use it to do something else entirely.

For example:

- The `with` function calls its lambda one time and returns the resulting value.
- The `map` function calls its lambda multiple times and returns an array of every resulting value.
- The `each` function throws away its lambda's values and returns a copy of its main argument.

Every lambda creates its own local scope which is anonymous, and contains variables which can not be accessed by qualified names from any other scope. The parent scope of a lambda is the local scope in which that lambda is written. When a lambda is written inside a class definition, its code block accesses local variables from that class, as well as variables from that class's ancestor scopes, and from the top scope. Lambdas can contain other lambdas, which makes the outer lambda the parent scope of the inner one.

A lambda is a value with the Callable data type, and functions using the modern function API (`Puppet::Functions`) use that data type to validate any lambda values it receives. However, the Puppet language doesn't provide any way to store or interact with Callable values except as lambdas provided to a function.

## Resource default statements

Resource default statements enable you to set default attribute values for a given resource type. Resource declarations within the area of effect that omits those attributes inherit the default values.

### Syntax

```
Exec {
  path        => '/usr/bin:/bin:/usr/sbin:/sbin',
  environment => 'RUBYLIB=/opt/puppetlabs/puppet/lib/ruby/site_ruby/2.1.0/',
  logoutput   => true,
  timeout     => 180,
}
```

The general form of resource defaults is:

- The capitalized resource type name. If the resource type name has a namespace separator (`::`), every segment must be capitalized, for example `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, each resource type that omits a given attribute uses that attribute's default value.

**Important:** Attributes set explicitly in a resource declaration override any default value.

Resource defaults are evaluation order dependent. Defaults are assigned to a created resource when a resource expression is evaluated; that is, when it is declared for inclusion in the catalog. Puppet uses the default values that are in effect for the type at evaluation.

Puppet uses dynamic scoping for resource defaults, even though it no longer uses dynamic variable lookup. This means that when you use a resource default statement in a class, it could affect any classes or defined types that class declares. Therefore, they should not be set outside of `site.pp`. You should use per-resource default attributes when possible.

Resource defaults declared in the local scope override any defaults received from parent scopes. Overriding of resource defaults is per attribute, not per block of attributes. This means local and parent resource defaults that don't conflict with each other are merged together.

## Resource collectors

Resource collectors select a group of resources by searching the attributes of each resource in the catalog, even resources which haven't yet been declared at the time the collector is written. Collectors realize virtual resources, are used in chaining statements, and override resource attributes. Collectors have an irregular syntax that enables them to function as a statement and a value.

### Syntax

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

The general form of a resource collector is:

- A capitalized resource type name. This cannot be `Class`, and there is no way to collect classes.

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

**Note:** Exported resource collectors have a slightly different syntax; see below.

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

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

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

Use parentheses to improve readability, and to modify the priority and grouping of `and` and `or` operators. You can create complex expressions using four operators.

**`==` (equality search)**

This operator is non-symmetric:

- 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 of Puppet.

For a given resource, this operator matches 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-symmetric:

- 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 of Puppet.

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

**Note:** This operator always match if the attribute's value is an array.

**`and`**

Both operands must be valid search expressions. For a given resource, this operator matches if both of the operands match for that resource. This operator has higher priority than `or`.

**`or`**

Both operands must be valid search expressions. For a given resource, this operator matches if either of the operands match for that resource. This operator has lower priority than `and`.

### Location

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

Collectors *cannot* be used in the following contexts:

- As the value of a resource attribute
- As the argument of a function
- Within an array or hash
- As the operand of an expression other than a chaining statement

### Behavior

A resource collector realizes any virtual resources matching its search expression. Empty search expressions match every resource of the specified resource type.

**Note:** A collector also collects and realizes any exported resources from the current node. When you use exported resources that you don't want realized, exclude them from the collector's search expression.

Collectors function as a value in two places:

- In a chaining statement, a collector acts as a proxy for every resource (virtual or not) that matches its search expression.
- When given a block of attributes and values, a collector sets and overrides those attributes for every resource (virtual or not) matching its search expression.

**Note:** Collectors used as values also realize any matching virtual resources. When you use virtualized resources, be careful when chaining collectors or using them for overrides.

### Exported resource collectors

An exported resource collector uses a modified syntax that realizes exported resources and imports resources published by other nodes.

To use exported resource collectors, enable catalog storage and searching (`storeconfigs`). See Exported resources for more details. To enable exported resources, follow the installation instructions and Puppet configuration instructions in the PuppetDB docs.

Like normal collectors, use exported resource collectors with attribute blocks and chaining statements.

**Note:** The search for exported resources also searches the catalog being compiled, to avoid having to perform an additional run before finding them in the store of exported resources.

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

## Virtual resources

A virtual resource declaration specifies a desired state for a resource without enforcing that state. Puppet manages the resource by realizing it elsewhere in your manifests. This divides the work done by a normal resource declaration into two steps. Although virtual resources are declared one time, they can be realized any number of times, similar to a class.

### Purpose

Virtual resources are useful for:

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

Because they both offer a safe way to add a resource to the catalog in multiple locations, virtual resources can be used in some of the same situations as classes. The features that distinguish virtual resources are:

- Searchability via resource collectors, which helps to 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.

### Syntax

Virtual resources are used in two steps: declaring and realizing. In this example, the `apache` class declares a virtual resource, and both the `wordpress` and `freight` classes realize it. The resource is managed on any node that has the `wordpress` or `freight` classes applied to it.

Declare: `modules/apache/manifests/init.pp`

```
@a2mod { 'rewrite':
  ensure => present,
} # note: The a2mod resource type is from the puppetlabs-apache module.
```

Realize: `modules/wordpress/manifests/init.pp`

```
realize A2mod['rewrite']
```

Realize again: `modules/freight/manifests/init.pp`

```
realize A2mod['rewrite']
```

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

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

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'])
```

**Note:** The realize function can be used multiple times on the same virtual resource and the resource is managed only one time.

A resource collector realizes any virtual resources that match its search expression:

```
User <| tag == web |>
```

If multiple resource collectors match a given virtual resource, Puppet manages only that resource one time.

**Note:** A collector also collects and realizes any exported resources from the current node. If you use exported resources that you don't want realized, take care to exclude them from the collector's search expression. Also, a collector used in an override block or a chaining statement also realizes any matching virtual resources.

### Behavior

A virtual resource declaration does not manage the state of a resource. Instead, it makes a virtual resource available to resource collectors and the `realize` function. When a resource is realized, Puppet manages its state.

Unrealized virtual resources are included in the catalog, but are marked inactive.

**Note:** Virtual resources do not depend on evaluation order. You can realize a virtual resource before the resource has been declared.

**CAUTION:** The realize function causes a compilation failure when attempting to realize a virtual resource that has not been declared. Resource collectors fail silently when they do not match any resources.

When 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 |>
```

**Note:** You can declare virtual resources of defined resource types. This causes every resource contained in the defined resource to behave virtually — they are not managed unless their virtual container is realized.

Virtual resources are evaluated in the run stage in which they are declared, not the run stage in which they are realized.

## Exported resources

An exported resource declaration specifies a desired state for a resource, and publishes the resource for use by other nodes. It does not manage the resource on the target system. Any node, including the node that exports it, can collect the exported resource and manage its own copy of it.

### Purpose

Exported resources enable the Puppet compiler to share information among nodes by combining information from multiple nodes' catalogs. This helps manage things that rely on nodes knowing the states or activity of other nodes.

**Note:** Exported resources rely on the compiler accessing the information, and can not use information that's never sent to the compiler, such as the contents of arbitrary files on a node.

The common use cases are monitoring and backups. A class that manages a service like PostgreSQL, exports a `nagios_service` resource which describes how to monitor the service, including information such as its hostname and port. The Nagios server collects every `nagios_service` resource, and automatically starts monitoring the Postgres server.

**Note:** Exported resources require catalog storage and searching (`storeconfigs`) enabled on your Puppet master. Both the catalog storage and the searchin, among other features, are provided by PuppetDB. To enable exported resources, see:

- Install PuppetDB on a server at your site
- Connect your Puppet master to PuppetDB

### Syntax

Using exported resources requires two steps: declaring and collecting. In the following examples, every node with the `ssh` class exports its own SSH host key and then collects the SSH host key of every node (including its own). This causes every node in the site to trust SSH connections from every other node.

```
class ssh {
  # Declare:
  @@sshkey { $::hostname:
    type => dsa,
    key  => $::sshdsakey,
  }
  # Collect:
  Sshkey <<| |>>
     }
```

To declare an exported resource, prepend @@ to the resource 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],
}
```

To collect exported resources, use an exported resource collector. Collect all exported `nagios_service` resources:

```
Nagios_service <<| |>>
```

This example, taken from puppetlabs-bacula, uses the ripienaar-concat module, collects exported file fragments for building a Bacula config file:

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

**Tip:** It's difficult to predict the title of an exported resource, because any node could be exporting it. It's best to search on a more general attribute, and this is one of the main use cases for tags.

For more information on the collector syntax and search expressions, see Exported resource collectors.

### Behavior

When catalog storage and searching (`storeconfigs`) are enabled, the master sends a copy of every compiled catalog to PuppetDB. PuppetDB retains the recent catalog for every node and provides the master with a search interface to each catalog.

Declaring an exported resource adds the resource to the catalog marked with an *exported* flag. Unless it was collected, this prevents the agent from managing the resource. When PuppetDB receives the catalog, it takes note of this flag.

Collecting an exported resource causes the master to send a search query to PuppetDB. PuppetDB responds with every exported resource that matches the search expression, and the master adds those resources to the catalog.

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 might not happen immediately. The master must have compiled a given node's catalog at least one time before its resources become available. When the 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 might be a delay between a node's catalog being compiled and its resources becoming available.

Normally, exported resource types include their default attribute values. However, defined types are evaluated for the catalog after the resource is collected, so their default values are not exported. To make sure a defined type's values are exported, set them explicitly.

To remove stale exported resources, expire or deactive the node that exported them. This ensures that any resources exported by that node stop appearing in the catalogs served to the remaining agent nodes. For details, see the documentation for deactivating or expiring nodes.

**CAUTION:** Each exported resource must be globally unique across every single node. If two nodes export resources with the same title or same name/namevar, the compilation fails when you attempt to collect both. Some pre-1.0 versions of PuppetDB do not fail in this case. To ensure uniqueness, every resource you export must 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.

**Restriction:** Exported resource collectors do not collect normal or virtual resources. They cannot retrieve non-exported resources from other nodes' catalogs.

The following example shows Puppet native types for managing Nagios configuration files. These types become powerful when you export and collect them.

For example, to create a class for Apache that adds a service definition on your Nagios host, automatically monitoring the web server:

`/etc/puppetlabs/puppet/modules/nagios/manifests/target/apache.pp`:

```
class nagios::target::apache {
  @@nagios_host { $::fqdn:
    ensure  => present,
    alias   => $::hostname,
    address => $::ipaddress,
    use     => 'generic-host',
  }
  @@nagios_service { "check_ping_${::hostname}":
    check_command       => 'check_ping!100.0,20%!500.0,60%',
    use                 => 'generic-service',
    host_name           => $::fqdn,
    notification_period => '24x7',
    service_description => "${::hostname}_check_ping"
  }
}
```

`/etc/puppetlabs/puppet/modules/nagios/manifests/monitor.pp`:

```
class nagios::monitor {
  package { [ 'nagios', 'nagios-plugins' ]: ensure => installed, }
  service { 'nagios':
    ensure     => running,
    enable     => true,
    #subscribe => File[$nagios_cfgdir],
    require    => Package['nagios'],
  }
```

Collect resources and populate `/etc/nagios/nagios_*.cfg`:

```
 Nagios_host <<||>>
  Nagios_service <<||>>
}
```

## Tags

Tags are useful for collecting resources, analyzing reports, and restricting catalog runs. Resources, classes, and defined type instances can have multiple tags associated with them, and they receive some tags automatically.

### Tag names

For information about the characters allowed in tag names, see reserved words and acceptable names.

### Assigning tags to resources

Every resource automatically receives the following tags:

- Its resource type.
- The full name of the class or defined type in which the resource was declared.
- Every namespace segment of the resource's class or defined type.

For example, a file resource in class `apache::ssl` is automatically assigned the tags `file`, `apache::ssl`, `apache`, and `ssl`. Do not manually assign tags with names that are the same as these automatically assigned tags.

**Tip:** Class tags are useful when setting up the tagmail module or testing refactored manifests.

Similar to relationships and most metaparameters, tags are passed along by containment. This means a resource receives all of the tags from the class and/or defined type that contains it. In the case of nested containment (a class that declares a defined resource, or a defined type that declares other defined resources), a resource receives tags from all of its containers.

The `tag` metaparameter accepts a single tag or an array, and these are added to the tags the resource already has. A `tag` can also be used with normal resources, defined resources, and classes (when using the resource-like declaration syntax).

Because containment applies to tags, the example below assigns the `us_mirror1` and `us_mirror2` tags to every resource contained by `Apache::Vhost['docs.puppetlabs.com']`.

To add multiple tags, use the tag metaparameter in a resource declaration:

```
apache::vhost {'docs.puppetlabs.com':
  port => 80,
  tag  => ['us_mirror1', 'us_mirror2'],
}
```

To assign tags to the surrounding container and all of the resources it contains, use the `tag` function inside a class definition or defined type. The example below assigns 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.

```
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': }
}
```

### Using tags

**Tip:** Tags are useful when used as an attribute in the search expression of a resource collector for realizing virtual and exported resources.

Puppet agent and Puppet apply use the `tags` setting to apply a subset of the node's catalog. This is useful when refactoring modules, and enables you to apply a single class on a test node.

The `tags` setting can be set in `puppet.conf` to restrict the catalog, or on the command line to temporarily restrict it. The value of the `tags` setting should be a comma-separated list of tags, with no spaces between tags:

```
$ sudo puppet agent --test --tags apache,us_mirror1
```

The tagmail module sends emails to arbitrary email addresses whenever resources with certain tags are changed.

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.

For more information, see:

- Processing reports
- Report format

## Run stages

Run stages are an additional way to order resources. Groups of classes run before or after everything else, without having to explicitly create relationships with other classes. The run stage feature has two parts: a `stage` resource type, and a `stage` metaparameter, which assigns a class to a named run stage.

### 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 are specified with metaparameters or with chaining arrows.

In this example, all classes assigned to the `first` stage are applied before the classes associated with the `main` stage, and both of those stages are applied before the `last` stage.

```
stage { 'first':
  before => Stage['main'],
}
stage { 'last': }
Stage['main'] -> Stage['last']
```

### Assigning classes to stages

After stages have been declared, use the `stage` metaparameter to assign a class to a custom stage.

This example ensures that the `apt-keys` class happens before all other classes, which is useful if most of your package resources rely on those keys.

```
class { 'apt-keys':
  stage => first,
}
```

### Limitations

Run stages have these limitations:

- To assign a class to a stage, you must use the resource-like class declaration syntax and supply the stage explicitly. You cannot assign classes to stages with the `include` function, or by relying on automatic parameter lookup from hiera while using resource-like class declarations.
- You cannot subscribe to or notify resources across a stage boundary.
- Classes that contain other classes, with either the `contain` function or the anchor pattern, can sometimes behave badly if declared with a run stage. If the contained class is declared only by its container, it works fine, but if it's declared anywhere outside its container, it often creates a dependency cycle that prevents the involved classes being applied.

**CAUTION:** Due to these limitations, use stages with the simplest of classes, and only when absolutely necessary. A valid use case is mass dependencies like package repositories.

## Details of complex behaviors

Within Puppet language there are complex behavior patterns regarding classes, defined types, and specific areas of code called scopes.

- Containment of resources on page 367
Containment enables you to control where and when specific parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types.
- Scope on page 369
A scope is a specific area of code that is partially isolated from other areas of code.
- Namespaces and autoloading on page 374
Class and defined type names can be broken up into segments called namespaces which enable the autoloader to find the class or defined type in your modules.

### Containment of resources

Containment enables you to control where and when specific parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types.

#### Containment

Classes and defined type instances *contain* the resources they declare. Contained resources are not applied before the container begins, and they finish before the container finishes.

This means that if any resource or class forms a relationship with the container, it forms the same relationship with every resource inside the container.

Consider this example:

```
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'],
}
```

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

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

### Containing resources

Resources of both native and defined resource types are automatically contained by the class or defined type in which they are declared.

### Containing classes

Having classes contain other classes can be useful, especially in larger modules where you want to improve code readability by moving chunks of implementation into separate files.

Unlike resources, Puppet does not automatically contain classes when they are declared inside another class. This is because classes can be declared in several places using `include` and similar functions. Most of these places shouldn't contain the class, and trying to contain it everywhere would cause huge problems. Instead, you must manually contain any classes that need to be contained.

#### The `contain` function

When a class should be contained, and if it isn't already declared, use the `contain` function to declare the class with include-like behavior, causing it to be contained by the surrounding class.

In this NTP module example, the service configuration is moved out into its own class:

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

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

This ensures that the exec happens after all the resources in both class `ntp` and class `ntp::service`. If `ntp::service` had been declared with `include` instead of `contain`, the exec would happen after the file and the package, but wouldn't necessarily happen after the service.

To contain classes that are declared with the resource-like declaration syntax, use the `contain` function after declaring the class:

```
class ntp {
  # ...
  class { 'ntp::service':
    enable => true,
  }
  contain 'ntp::service'
  # ...
}
```

#### Anchor pattern containment (for compatibility with Puppet version 3.4.0 and earlier)

Versions prior to Puppet 3.4.0 and Puppet Enterprise 3.2 do not ship with the `contain` function. If your code needs to support these versions, it should contain classes with the anchor pattern.

**Note:** To use the anchor pattern, install the `puppetlabs/stdlib` module. which includes the dummy `anchor` resource type.

To use the anchor pattern:

- The containing class must include two uniquely-named `anchor` resources, which are resources that don't have any effect on the target system, and only exist to form relationships with.
- Any contained classes must have relationships ensuring they happen after one anchor and before the other.

In this NTP module example, the service configuration is moved out into its own class:

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

  # roughly equivalent to "contain ntp::service":
  anchor { 'ntp_first': } -> Class['ntp::service'] -> anchor { 'ntp_last': }

  package { 'ntp':
    ...
  }
}

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

In this case, the `ntp::service` class behaves like it's contained by the `ntp` class. Resources like the timestamp `exec` can form relationships with the `ntp` class and be assured that no relevant resources float out of order.

## Scope

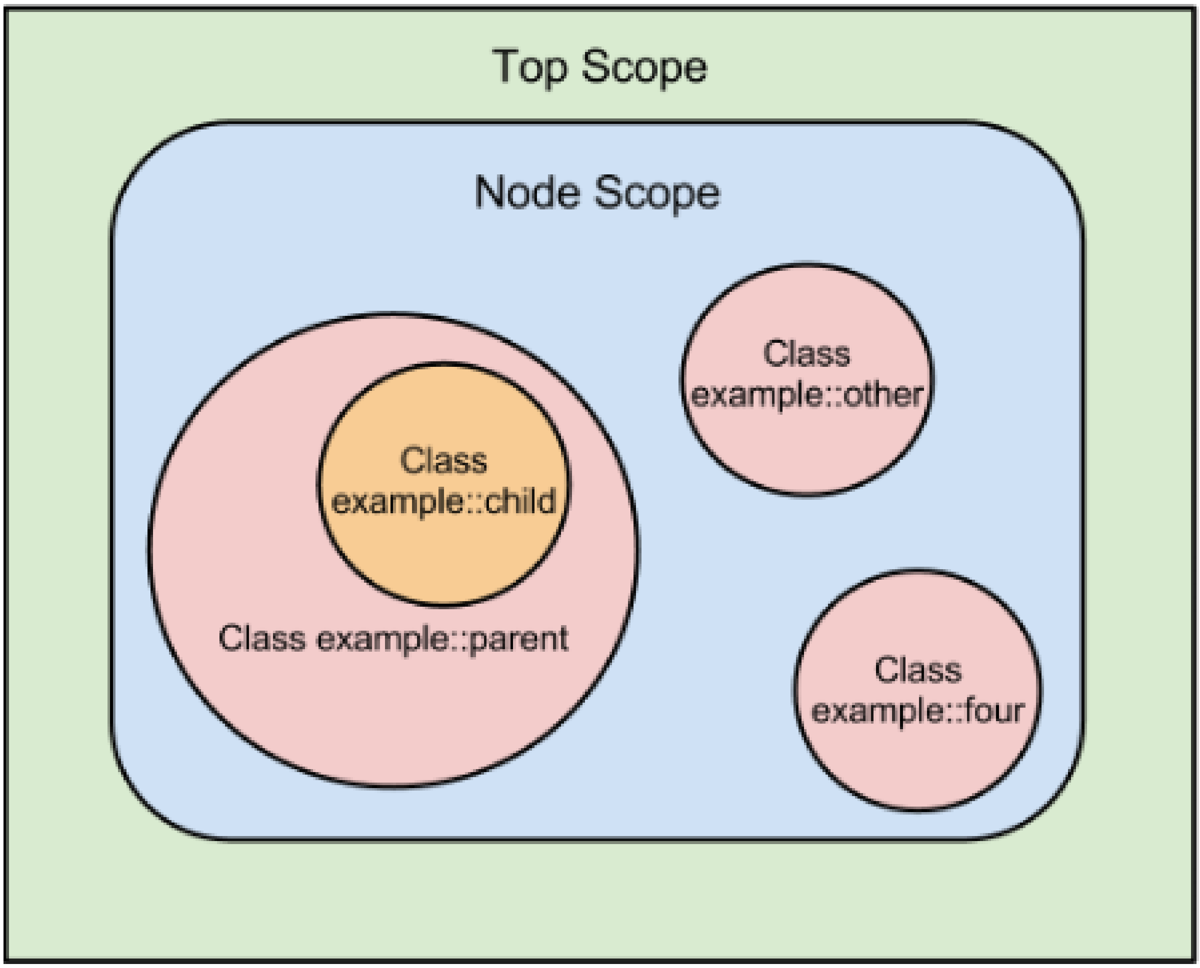
A scope is a specific area of code that is partially isolated from other areas of code.

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.




A particular scope has access to its own contents, and also receives additional contents from its parent scope, node scope, and top scope. The rules for how Puppet determines a local scope's parent are described in scope lookup rules.

In the diagram above:

- Top scope can access variables and defaults only 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, the `example::parent` 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. Only one node scope can exist at a time because only one node definition can match a given node.

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.

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

```
# 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:
```

#### Local scopes

Code inside a class definition, defined type, or lambda 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. For more information, see scope lookup rules.

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

```
# /etc/puppetlabs/code/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/puppetlabs/code/environments/production/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:
```

### 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. If multiple variables with the same name are available, Puppet uses the most local one.

```
# /etc/puppetlabs/code/modules/scope_example/manifests/init.pp
class scope_example {
  $variable = "Hi, I'm local!"
  notify {"Message from here: $variable":}
}

# /etc/puppetlabs/code/environments/production/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 are merged, and only the attributes that conflict are overridden.

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

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

  file {'/tmp/example':}
}

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

include scope_example
```

### Scope of external node classifier data

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

This results the most expected behavior: variables from an ENC are available everywhere, and classes can use node-specific variables.

**Note:** This means compilation fails if a site manifest tries to set a variable that was already set at top scope by an ENC.

Node scope only exists if there is at least one node definition in the main manifest. If no node definitions exist, then ENC classes get declared at top scope.

### 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.

Node scope and the local scopes created by lambdas and 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 *qualified global name*.

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

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

In the following example, the variable `$local_copy` is set to the value of the `$confdir` variable from the `apache::params` class:

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

**Note:**

A class must be declared to access its variables; just having the class available in your modules is insufficient.

This means the availability of out-of-scope variables is evaluation-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.

Because the top scope's name is the empty string, `$::my_variable` refers to the top-scope value of `$my_variable`, even if `$my_variable` has a different value in local scope.

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

### Scope lookup rules

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

There are two sets of scope lookup rules: static scope and dynamic scope. Puppet uses:

- Static scope for variables.
- Dynamic scope for resource defaults.

### Static scope

In static scope, which Puppet uses for looking up variables, parent scopes are assigned in the following ways:

- Classes can receive parent scopes by class inheritance, using the `inherits` keyword. A derived class receives the contents of its base class in addition to the contents of node and top scope.
- A lambda's parent scope is the local scope in which the lambda is written. It can access variables in that scope by their short names.

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

**Note:** Static scope has the following characteristics:

- Scope contents are predictable and do not depend on evaluation order.
- Scope contents can be determined simply by looking at the relevant class definition; the place where a class or defined 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.

### Dynamic scope

In dynamic scope, which Puppet uses for looking up resource defaults, parent scopes are assigned by both inheritance and declaration, with preference given to inheritance. The full list of rules is:

- Each scope has only one parent, but can 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.

**Note:** 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 might have been declared.
- In some cases, you can only determine a scope's contents by executing the code.
- Because classes can be declared multiple times with the `include` function, the contents of a given scope are evaluation-order dependent.

## Namespaces and autoloading

Class and defined type names can be broken up into segments called namespaces which enable the autoloader to find the class or defined type in your modules.

### Syntax

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

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

### Autoloader behavior

When a class or defined resource is declared, Puppet uses its full name to find the class or defined type in your modules. 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.

Names map to file locations as follows:

- The first segment in a name, excluding the empty top namespace, identifies the module. If this is the only segment, the file name is `init.pp`.
- The last segment identifies the file name, minus the `.pp` extension.
- Any segments between the first and last are subdirectories under the `manifests` directory.

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

| Name | File path |
|---|---|
| `apache` | `<MODULE DIRECTORY>/apache/manifests/init.pp` |
| `apache::mod` | `<MODULE DIRECTORY>/apache/manifests/mod.pp` |

| Name | File path |
| --- | --- |
| `apache::mod::passenger` | `<MODULE DIRECTORY>/apache/manifests/mod/passenger.pp` |

**Note:** The `init.pp` file always contains a class or defined type with the same name as the module, and any other `.pp` file contains a class or defined type with at least two namespace segments. For example, `apache.pp` would contain a class named `apache::apache`. This means you can't name a class `<MODULE NAME>::init`.

### Nested definitions and missing files

If a class or defined type is defined inside another class or defined type definition, its name goes under the outer definition's namespace.

This causes its real name to be something other than the name it was defined with. For example, in the following code, the interior class's real name is `first::second`:

```
class first {
  class second {
    ...
  }
}
```

However, searching your code for that real name returns nothing. Also, it causes class `first::second` to be defined in the wrong file. **Avoid structuring your code like this.**

If the manifest file that corresponds to a name doesn't exist, Puppet continues to look for the requested class or defined type. It does this by removing the final segment of the name and trying to load the corresponding file, continuing to fall back until it reaches the module's `init.pp` file.

Puppet loads the first file it finds like this, and raises an error if that file doesn't contain the requested class or defined type.

This behavior allows you to put a class or defined type in the wrong file and still have it work. But structuring things this way is not recommended.

# Writing custom functions

Use the Puppet language, or the Ruby API to create custom functions.


- Custom functions on page 376

Puppet includes many built-in functions, and more are available in modules on the Forge. You can also write your own custom functions.

- Custom Puppet functions on page 377

You can write simple custom functions in the Puppet language, to transform data and construct values. A function can optionally take one or more parameters as arguments. A function returns a calculated value from its final expression.

- Writing custom functions in Ruby on page 381

You can write powerful and flexible functions using Ruby.

- Secrets store integrations and agent-side data retrieval on page 393

Puppet agents can fetch or calculate data for themselves at catalog application time. One use case for this is to securely retrieve sensitive information like passwords from a secrets store.

- Write a Puppet function to store secrets on page 394

Use the `Deferred` type to create a function that you add to a module to redact sensitive information.


## Custom functions

Puppet includes many built-in functions, and more are available in modules on the Forge. You can also write your own custom functions.

Functions are plugins used during catalog compilation. When a Puppet manifest calls a function, that function runs and returns a value. Most functions only produce values, but functions can also:

- Cause side effects that modify a catalog. For example, the `include` function adds classes to a catalog.
- Evaluate a provided block of Puppet code, using arguments to determine how that code runs.

Functions usually take one or more arguments, which determine the return value and the behavior of any side effects.

If you need to manipulate data or communicate with third-party services during catalog compilation, and if the built-in functions, or functions from Forge modules, aren't sufficient, you can write custom functions for Puppet.

Custom functions work just like Puppet's built-in functions: you can call them during catalog compilation to produce a value or cause side effects. You can use your custom functions locally, and you can share them with other users.

To make a custom function available to Puppet, you must put it in a module or an environment, in the specific locations where Puppet expects to find functions.

Puppet offers two interfaces for writing custom functions:

| Interface | Description |
|---|---|
| The Puppet language | To write functions in the Puppet language, you don't need to know any Ruby. However, it's less powerful than the Ruby API. Puppet functions can have only one signature per function, and can't take a lambda (a block of Puppet code) as an argument. |
| The Ruby functions API | The more powerful and flexible way to write functions. This method requires some knowledge of Ruby. You can use Ruby to write iterative functions. |

### Guidelines for writing custom functions

Whenever possible, avoid causing side effects. Side effects are any change other than producing a value, such as modifying the catalog by adding classes or resources to it.

In Ruby functions, it's possible to change the values of existing variables. Never do this, because Puppet relies on those variables staying the same.

### Documenting functions

For information about documenting your functions, see Puppet Strings .

Related topics: Function calls, Catalog compilation, Environments, Modules, Puppet Forge, Iterative functions.

## Custom Puppet functions

You can write simple custom functions in the Puppet language, to transform data and construct values. A function can optionally take one or more parameters as arguments. A function returns a calculated value from its final expression.

**Note:** While many functions can be written in the Puppet language, it doesn't support all of the same features as pure Ruby. For information about writing Ruby functions, which can perform more complex work, see Writing functions in Ruby. For information about iterative functions, which can be invoked by, but not written exclusively with, Puppet code, see Writing iterative functions.

### Syntax of functions

```
function <MODULE NAME>::<NAME>(<PARAMETER LIST>) >> <RETURN TYPE> {
  ... body of function ...
  final expression, which is the returned value of the function
}
```

The general form of a function written in Puppet language is:

- The keyword `function`.
- The namespace of the function. This should match the name of the module the function is contained in.
- The namespace separator, a double colon `::`
- The name of the function.
- An optional parameter list, which consists of:
  - An opening parenthesis `(`
  - A comma-separated list of parameters (for example, `String $myparam = "default value"`). Each parameter consists of:
    - An optional data type, which restricts the allowed values for the parameter (defaults to `Any`).
    - A variable name to represent the parameter, including the `$` prefix.
    - An optional equals sign `=` and default value, which must match the data type, if one was specified.
  - An optional trailing comma after the last parameter.
  - A closing parenthesis `)`
- An optional return type, which consists of:
  - Two greater-than signs `>>`
  - A data type that matches every value the function could return.
- An opening curly brace `{`
- A block of Puppet code, ending with an expression whose value is returned.
- A closing curly brace `}`

For example:

```
function apache::bool2http(Variant[String, Boolean] $arg) >> String {
  case $arg {
    false, undef, /(?i:false)/ : { 'Off' }
    true, /(?i:true)/          : { 'On' }
    default               : { "$arg" }
  }
}
```

### Order and optional parameters

Puppet passes arguments by parameter position. This means that the order of parameters is important. Parameter names do not affect the order in which they are passed.

If a parameter has a default value, then it's optional to pass a value for it when you're calling the function. If the caller doesn't pass in an argument for that parameter, the function uses the default value. However, because parameters are passed by position, when you write the function, you must list optional parameters after all required parameters. If you put a required parameter after an optional one, it causes an evaluation error.

### Variables in default parameters values

If you reference a variable as a default value for a parameter, Puppet starts looking for that variable at top scope. For example, if you use `$fqdn` as a variable, but then call the function from a class that overrides `$fqdn`, the parameter's default value is the value from top scope, not the value from the class. You can reference qualified variable names in a function default value, but compilation fails if that class isn't declared by the time the function is called.

### The extra arguments parameter

You can specify that a function's last parameter is an extra arguments parameter. The extra arguments parameter collects an unlimited number of extra arguments into an array. This is useful when you don't know in advance how many arguments the caller provides.

To specify that the parameter should collect extra arguments, start its name with an asterisk `*`, for example `* $others`. The asterisk is valid only for the last parameter.

**Tip:** An extra argument's parameter is always optional.

The value of an extra argument's parameter is an array, containing every argument in excess of the earlier parameters. You can give it a default value, which has some automatic array wrapping for convenience:

- If the provided default is a non-array value, the real default is a single-element array containing that value.
- If the provided default is an array, the real default is that array.

If there are no extra arguments and there is no default value, it's an empty array.

An extra arguments parameter can also have a data type. Puppet uses this data type to validate the elements of the array. That is, if you specify a data type of `String`, the real data type of the extra arguments parameter is `Array[String]`.

### Return types

Between the parameter list and the function body, you can use `>>` and a data type to specify the type of the values the function returns. For example, this function only returns strings:

```
function apache::bool2http(Variant[String, Boolean] $arg) >> String {
  ...
}
```

The return type serves two purposes:

- Documentation. Puppet Strings includes information about the return value of a function.
- Insurance. If something goes wrong and your function returns the wrong type (such as `undef` when a string is expected), it fails early with an informative error instead of allowing compilation to continue with an incorrect value.

### The function body

In the function body, put the code required to compute the return value you want, given the arguments passed in. Avoid declaring resources in the body of your function. If you want to create resources based on inputs, use defined types instead.

The final expression in the function body determines the value that the function returns when called. Most conditional expressions in the Puppet language have values that work in a similar way, so you can use an `if` statement or a `case` statement as the final expression to give different values based on different numbers or types of inputs. In the following example, the case statement serves as both the body of the function, and its final expression.

```
function apache::bool2http(Variant[String, Boolean] $arg) >> String {
  case $arg {
    false, undef, /(?i:false)/ : { 'Off' }
    true, /(?i:true)/          : { 'On' }
    default               : { "$arg" }
  }
}
```

### Locations

Store the functions you write in a module's `functions` folder, which is a top-level directory (a sibling of `manifests` and `lib`). Define only one function per file, and name the file to match the name of the function being defined. Puppet is automatically aware of functions in a valid module and autoloads them by name.

Avoid storing functions in the main manifest. Functions in the main manifest override any function of the same name in all modules (except built-in functions).

### Names

Give your function a name that clearly reveals what it does. For more information about names, including restrictions and reserved words, see Puppet naming conventions.

Related topics: Arrays, Classes, Data types, Conditional expressions, Defined types, Namespaces and autoloading, Variables.

### Calling a function

A call to a custom function behaves the same as a call to any built-in Puppet function, and resolves to the function's returned value.

After a function is written and available in a module where the autoloader can find it, you can call that function, either from a Puppet manifest that lists the containing module as a dependency, or from your main manifest.

Any arguments you pass to the function are mapped to the parameters defined in the function's definition. You must pass arguments for the mandatory parameters, but you can choose whether you want to pass in arguments for optional parameters.

Functions are autoloaded and are available to other modules unless those modules have specified dependencies. If a module has a list of dependencies in its `metadata.json` file, only custom functions from those specific dependencies are loaded.

Related topics: namespaces and autoloading, module metadata, main manifest directory

### Complex example of a function

The following code example is a re-written version of a Ruby function from the `postgresql` module into Puppet code. This function translates the IPv4 and IPv6 Access Control Lists (ACLs) format into a resource suitable for `create_resources`. In this case, the filename would be `acls_to_resource_hash.pp`, and it would be saved in a folder named `functions` in the top-level directory of the `postgresql` module.

```
function postgresql::acls_to_resource_hash(Array $acls, String $id, Integer
 $offset) {

  $func_name = "postgresql::acls_to_resources_hash()"

  # The final hash is constructed as an array of individual hashes
  # (using the map function), the result of that
  # gets merged at the end (using reduce).
  #
  $resources = $acls.map |$index, $acl| {
    $parts = $acl.split('\s+')
    unless $parts =~ Array[Data, 4] {
      fail("${func_name}: acl line $index does not have enough parts")
    }

    # build each entry in the final hash
    $resource = { "postgresql class generated rule ${id} ${index}" =>
      # The first part is the same for all entries
      {
        'type'     => $parts[0],
        'database' => $parts[1],
        'user'     => $parts[2],
        'order'    => sprintf("'%03d'", $offset + $index)
      }
      # The rest depends on if first part is 'local',
      # the length of the parts, and the value in $parts[4].
      # Using a deep matching case expression is a good way
      # to untangle if-then-else spaghetti.
      #
      # The conditional part is merged with the common part
      # using '+' and the case expression results in a hash
      #
      +
      case [$parts[0], $parts, $parts[4]] {

        ['local', Array[Data, 5], default] : {
          { 'auth_method' => $parts[3],
            'auth_option' => $parts[4, -1].join(" ")
          }
        }

        ['local', default, default] : {
          { 'auth_method' => $parts[3] }
        }

        [default, Array[Data, 7], /^\d/] : {
          { 'address'     => "${parts[3]} ${parts[4]}",
            'auth_method' => $parts[5],
            'auth_option' => $parts[6, -1].join(" ")
          }
        }

        [default, default, /^\d/] : {
          { 'address'     => "${parts[3]} ${parts[4]}",
            'auth_method' => $parts[5]
          }
```

```
        }

        [default, Array[Data, 6], default] : {
          { 'address'     => $parts[3],
            'auth_method' => $parts[4],
            'auth_option' => $parts[5, -1].join(" ")
          }
        }

        [default, default, default] : {
          { 'address'     => $parts[3],
            'auth_method' => $parts[4]
          }
        }
      }
    }
    $resource
  }
  # Merge the individual resource hashes into one
  $resources.reduce({}) |$result, $resource| { $result + $resource }
}
```

## Writing custom functions in Ruby

You can write powerful and flexible functions using Ruby.

- Overview on page 381
Get started with an overview of Ruby custom functions.
- Ruby function signatures on page 383
Functions can specify how many arguments they expect, and a data type for each argument. The rule set for a function's arguments is called a signature**.**
- Using special features in implementation methods on page 387
For the most part, implementation methods are normal Ruby. However, there are some special features available for accessing Puppet variables, working with provided blocks of Puppet code, and calling other functions.
- Iterative functions on page 389
You can use iterative types to write efficient iterative functions, or to chain together the iterative functions built into Puppet.
- Refactoring legacy 3.x functions on page 391
If you have Ruby functions written with the legacy 3.x API, refactor them to ensure that they work correctly with current versions of Puppet.

### Overview

Get started with an overview of Ruby custom functions.

#### Syntax of Ruby functions

To write a new function in Ruby, use the `Puppet::Functions.create_function` method. You don't need to `require` any Puppet libraries. Puppet handle libraries automatically when it loads the function.

```
Puppet::Functions.create_function(:<FUNCTION NAME>) do
  dispatch :<METHOD NAME> do
    param '<DATA TYPE>', :<ARGUMENT NAME (displayed in docs/errors)>
    ...
  end

  def <METHOD NAME>(<ARGUMENT NAME (for local use)>, ...)
    <IMPLEMENTATION>
  end
```

```
end
```

The `create_function` method requires:

- A function name.
- A block of code (which takes no arguments). This block contains:
  - One or more signatures to configure the function's arguments.
  - An implementation method for each signature. The return value of the implementation method is the return value of the function.

For example:

```
# /etc/puppetlabs/code/environments/production/modules/mymodule/lib/puppet/
functions/mymodule/upcase.rb
Puppet::Functions.create_function(:'mymodule::upcase') do
  dispatch :up do
    param 'String', :some_string
  end

  def up(some_string)
    some_string.upcase
  end
end
```

### Location

Place a Ruby function in its own file, in the `lib/puppet/functions` directory of either a module or an environment. The filename must match the name of the function, and have the `.rb` extension. For namespaced functions, each segment prior to the final one must be a subdirectory of `functions`, and the final segment must be the filename.

| Function name | File location |
|---|---|
| `upcase` | `<MODULES DIR>/mymodule/lib/puppet/functions/upcase.rb` |
| `upcase` | `/etc/puppetlabs/code/environments/production/lib/puppet/functions/upcase.rb` |
| `mymodule::upcase` | `<MODULES DIR>/mymodule/lib/puppet/functions/mymodule/upcase.rb` |
| `environment::upcase` | `/etc/puppetlabs/code/environments/production/lib/puppet/functions/environment/upcase.rb` |

Functions are autoloaded and made available to other modules unless those modules specify dependencies. After a function is written and available (in a module where the autoloader can find it), you can call that function in any Puppet manifest that lists the containing module as a dependency, and also from your main manifest. If a module has a list of dependencies in its `metadata.json` file, it loads custom functions only from those specific dependencies.

### Names

Function names are similar to class names. They consist of one or more segments. Each segment must start with a lowercase letter, and can include:

- Lowercase letters
- Numbers
- Underscores

If a name has multiple segments, separate them with a double-colon (`::`) namespace separator.

Match each segment with this regular expression:

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

Match the full name with this regular expression:

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

Function names can be either global or namespaced:

- Global names have only one segment (`str2bool`), and can be used in any module or environment. Global names are shorter, but they're not guaranteed to be unique — if you use a function name that is already in use by another module, Puppet might load the wrong module when you call it.
- Namespaced names have multiple segments (`stdlib::str2bool`), and are guaranteed to be unique. The first segment is dictated by the function's location:
  - In an environment, use `environment` (`environment::str2bool`).
  - In a module, use the module's name (`stdlib::str2bool` for a function stored in the `stdlib` module).

Most functions have two name segments, although it's legal to use more.

Examples of legal function names:

- `num2bool` (a function that could come from anywhere)
- `postgresql::acls_to_resource_hash` (a function in the `postgresql` module)
- `environment::hash_from_api_call` (a function in an environment)

Examples of illegal function names:

- `6_pack` (must start with a letter)
- `_hash_from_api_call` (must start with a letter)
- `Find-Resource` (can only contain lowercase letters, numbers, and underscores)

### Passing names to `create_function` as symbols

When you call the `Puppet::Functions.create_function` method, pass the function's name to it as a Ruby symbol.

To turn a function name into a symbol:

- If the name is global, prefix it with a colon (`:str2bool`).
- If it's namespaced: put the name in quotation marks, and prefix the full quoted string with a colon (`:'stdlib::str2bool'`).

Related topics: Puppet modules, Environments, Main manifest, Module metadata, Ruby symbols.

## Ruby function signatures

Functions can specify how many arguments they expect, and a data type for each argument. The rule set for a function's arguments is called a signature**.**

Because Puppet functions support more advanced argument checking than Ruby does, the Ruby functions API uses a lightweight domain-specific language (DSL) to specify signatures.

Ruby functions can have multiple signatures. Using multiple signatures is an easy way to have a function behave differently when passed by different types or quantities of arguments. Instead of writing complex logic to decide what to do, you can write separate implementations and let Puppet select the correct signature.

If a function has multiple signatures, Puppet uses its data type system to check each signature in order, comparing the allowed arguments to the arguments that were actually passed. As soon as Puppet finds a signature that can accept the provided arguments, it calls the associated implementation method, passing the arguments to that method. When the method finishes running and returns a value, Puppet uses that as the function's return value. If none of the function's signatures match the provided arguments, Puppet fails compilation and logs an error message describing the mismatch between the provided and expected arguments.

### Conversion of Puppet and Ruby data types

When function arguments are passed to a Ruby method, they're converted to Ruby objects. Similarly, when the Puppet manifest regains control, it converts the method's return value into a Puppet data type.

Puppet converts data types between the Puppet language and Ruby as follows:

| Puppet | Ruby |
|---|---|
| Boolean | `Boolean` |
| Undef | `NilClass` (value `nil`) |
| String | `String` |
| Number | subtype of `Numeric` |
| Array | `Array` |
| Hash | `Hash` |
| Default | `Symbol` (value `:default`) |
| Regexp | `Regexp` |
| Resource reference | `Puppet::Pops::Types::PResourceType, or Puppet::Pops::Types::PHostClassType` |
| Lambda (code block) | `Puppet::Pops::Evaluator::Closure` |
| Data type (`Type`) | A type class under `Puppet::Pops::Types`. For example, `Puppet::Pops::Types::PIntegerType` |

**Tip:** When writing iterative functions, use iterative types instead of Puppet types.

### Writing signatures with `dispatch`

To write a signature, use the `dispatch` method.

The `dispatch` method takes:

- The name of an implementation method, provided as a Ruby symbol. The corresponding method must be defined somewhere in the `create_function` block, usually after all the signatures.
- A block of code which only contains calls to the parameter and return methods.

```
# A signature that takes a single string argument
  dispatch :camelcase do
    param 'String', :input_string
    return_type 'String' # optional
  end
```

### Using parameter methods

In the code block of a `dispatch` statement, you can specify arguments with special parameter methods. All of these methods take two arguments:

- The allowed data type for the argument, as a string. Types are specified using Puppet's data type syntax.
- A user-facing name for the argument, as a symbol. This name is only used in documentation and error messages; it doesn't have to match the argument names in the implementation method.

The order in which you call these methods is important: the function's first argument goes first, followed by the second, and so on. The following parameter methods are available:

| Model name | Description |
|---|---|
| `param` or `required_param` | A mandatory argument. You can use any number of these.<br><br>**Position**: All mandatory arguments must come first. |
| `optional_param` | An argument that can be omitted. You can use any number of these. When there are multiple optional arguments, users can only pass latter ones if they also provide values for the prior ones. This also applies to repeated arguments.<br><br>**Position**: Must come after any required arguments. |
| `repeated_param` or `optional_repeated_param` | A repeatable argument, which can receive zero or more values. A signature can only use one repeatable argument.<br><br>**Position**: Must come after any non-repeating arguments. |
| `required_repeated_param` | A repeatable argument, which must receive one or more values. A signature can only use one repeatable argument.<br><br>**Position**: Must come after any non-repeating arguments. |
| `block_param` or `required_block_param` | A mandatory lambda (block of Puppet code). A signature can only use one block.<br><br>**Position**: Must come after all other arguments. |
| `optional_block_param` | An optional lambda. A signature can only use one block.<br><br>**Position**: Must come after all other arguments. |

When specifying a repeatable argument, note that:

- In your implementation method, the repeatable argument appears as an array, which contains all the provided values that weren't assigned to earlier, non-repeatable arguments.
- The specified data type is matched against each value for the repeatable argument, not the repeatable argument as a whole. For example, if you want to accept any number of numbers, specify `repeated_param 'Numeric', :values_to_average, not repeated_param 'Array[Numeric]', :values_to_average`.

For lambdas, note that:

- The data type for a block argument is `Callable`, or a `Variant` that only contains `Callables`.
- The `Callable` type can optionally specify the type and quantity of parameters that the lambda accepts. For example, `Callable[String, String]` matches any lambda that can be called with a pair of strings.

### Matching arguments with implementation methods

The implementation method that corresponds to a signature must be able to accept any combination of arguments that the signature might allow.

If the signature has optional arguments, the corresponding method arguments need default values. Otherwise, the function fails if the arguments are omitted. For example:

```
dispatch :epp do
  required_param 'String', :template_file
  optional_param 'Hash', :parameters_hash
end

def epp(template_file, parameters_hash = {})
  # Note that parameters_hash defaults to an empty hash.
end
```

If the signature has a repeatable argument, the method must use a splat parameter (`*args`) as its final argument. For example:

```
dispatch :average do
  required_repeated_param 'Numeric', :values_to_average
end

def average(*values)
  # Inside the method, the `values` variable is an array of numbers.
end
```

### Using the `return_type` method

After specifying a signature's arguments, you can use the `return_type` method to specify the data type of its return value. This method takes one argument: a Puppet data type, specified as a string.

```
dispatch :camelcase do
  param 'String', :input_string
  return_type 'String'
end
```

The return type serves two purposes: documentation, and insurance.

- Puppet Strings can include information about the return value of a function.
- If something goes wrong and your function returns the wrong type (like `nil` when a string is expected), it fails early with an informative error instead of allowing compilation to continue with an incorrect value.

### Specifying aliases using `local_types`

If you're using complicated abstract data types to validate arguments, and you're using these data types in multiple signatures, they can become difficult to work with and maintain. In these cases, you can specify short aliases for your complex data types and use the aliases in your signatures.

To specify aliases, use the `local_types` method:

- You must call `local_types` only one time, before any signatures.
- The `local_types` method takes a lambda, which only contains calls to the `type` method.
- The `type` method takes a single string argument, in the form `'<NAME> = <TYPE>'`.
    - Capitalize the name, camel case word (`PartColor`), similar to a Ruby class name or the existing Puppet data types.
    - The type is a valid Puppet data type.

Example:

```
local_types do
  type 'PartColor = Enum[blue, red, green, mauve, teal, white, pine]'
  type 'Part = Enum[cubicle_wall, chair, wall, desk, carpet]'
  type 'PartToColorMap = Hash[Part, PartColor]'
```

```
end

dispatch :define_colors do
  param 'PartToColorMap', :part_color_map
end

def define_colors(part_color_map)
  # etc
end
```

### Using automatic signatures

If your function only needs one signature, and you're willing to skip the API's data type checking, you can use an automatic signature. Be aware that there are some drawbacks to using automatic signatures.

Although functions with automatic signatures are simpler to write, they give worse error messages when called incorrectly. You'll get a useful error if you call the function with the wrong number of arguments, but if you give the wrong type of argument, you'll get something unhelpful. For example, if you pass the function above a number instead of a string, it reports `Error: Evaluation Error: Error while evaluating a Function Call, undefined method 'split' for 5:Fixnum at /Users/nick/Desktop/test2.pp:7:8 on node magpie.lan.`

If it's possible that your function will be used by anyone other than yourself, support your users by writing a signature with `dispatch`.

To use an automatic signature:

- Do not write a `dispatch` block.
- Define one implementation method whose name matches the final namespace segment of the function's name.

```
Puppet::Functions.create_function(:'stdlib::camelcase') do
  def camelcase(str)
    str.split('_').map{|e| e.capitalize}.join
  end
end
```

In this case, because the last segment of `stdlib::camelcase` is `camelcase`, we must define a method named `camelcase`.

Related topics: Ruby symbols, Abstract data types.

## Using special features in implementation methods

For the most part, implementation methods are normal Ruby. However, there are some special features available for accessing Puppet variables, working with provided blocks of Puppet code, and calling other functions.

### Accessing Puppet variables

We recommend that most functions only use the arguments they are passed. However, you also have the option of accessing globally-reachable Puppet variables. The main use case for this is accessing facts, trusted data, or server data.

**Remember:** Functions cannot access local variables in the scope from which they were called. They can only access global variables or fully-qualified class variables.

To access variables, use the special `closure_scope` method, which takes no arguments and returns a `Puppet::Parser::Scope` object.

Use `#[](varname)` to call on a scope object, which returns the value of the specified variable. Make sure to exclude the `$` from the variable name. For example:

```
Puppet::Functions.create_function(:'mymodule::fqdn_rand') do
```

```
  dispatch :fqdn do
    # no arguments
  end

  def fqdn()
    scope = closure_scope
    fqdn = scope['facts']['networking']['fqdn']
    # ...
  end
end
```

### Working with lambdas (code blocks)

If their signatures allow it, functions can accept lambdas (blocks of Puppet code). If a function has a lambda, it generally needs to execute it. To do this, use Ruby's normal block calling conventions.

| Convention | Description |
| --- | --- |
| `block_given?` | If your signature specified an optional code block, your implementation method can check for its presence with the `block_given?` method. This is `true` if a block was provided, `false` if not. |
| `yield()` | When you know a block was provided, you can execute it any number of times with the `yield()` method.<br><br>The arguments to `yield` are passed as arguments to the lambda. If your signature specified the number and type of arguments the lambda expects, you can call it with confidence. The return value of the `yield` call is the return value of the provided lambda. |

If you need to introspect a provided lambda, or pass it on to some other method, an implementation method can capture it as a `Proc` by specifying an extra argument with an ampersand (&) flag. This works the same way as capturing a Ruby block as a `Proc`. After you capture the block, you can execute it with `#call` instead of `yield`. You can also use any other `Proc` instance methods to examine it.

```
def implementation(arg1, arg2, *splat_arg, &block)
  # Now the `block` variable has the provided lambda, as a Proc.
  block.call(arg1, arg2, splat_arg)
end
```

### Calling other functions

If you want to call another Puppet function (like `include`) from inside a function, use the special `call_function(name, *args, &block)` method.

```
# Flatten an array of arrays of strings, then pass it to include:
def include_nested(array_of_arrays)
  call_function('include', *array_of_arrays.flatten)
end
```

- The first argument must be the name of the function to call, as a string.
- The next arguments can be any data type that the called function accepts. They are passed as arguments to the called function.

- The last argument can be a Ruby `Proc`, or a Puppet lambda previously captured as a `Proc`. You can also provide a block of Ruby code using the normal block syntax.

```
def my_function1(a, b, &block)
  # passing given Proc
  call_function('my_other_function', a, b, &block)
end

def my_function2(a, b)
  # using a Ruby block
  call_function('my_other_function', a, b) { |x| ... }
end
```

Related topics: `Proc`, `yield`, `block_given?`, Puppet variables, Lambdas, Facts and built-in variables.

## Iterative functions

You can use iterative types to write efficient iterative functions, or to chain together the iterative functions built into Puppet.

Iterative functions include `Iterable` and `Iterator` types, as well as other types you can iterate over, such as arrays and hashes. For example, an `Array[Integer]` is also an `Iterable[Integer]`.

**Tip:** `Iterable` and `Iterator` types are used internally by Puppet to efficiently chain the results of its built-in iterative functions. You can't write iterative functions solely in the Puppet language. For help writing less complex functions in Puppet code, see Writing functions in Puppet.

### `Iterable` and `Iterator` type design

The `Iterable` type represents all things an iterative function can iterate over. Before this type was introduced in Puppet 4.4, if you wanted to design working iterative functions, you'd have to write code that accommodated all relevant types, such as `Array`, `Hash`, `Integer`, and `Type[Integer]`.

Signatures of iterative functions accept an `Iterable` type argument. This means that you no longer have to design iterative functions to check against every type. This behavior does not affect how the Puppet code that invokes these functions works, but does change the errors you see if you try to iterate over a value that does not have the `Iterable` type.

The `Iterator` type, which is a subtype of `Iterable`, is a special algorithm-based `Iterable` not backed by a concrete data type. When asked to produce a value, an `Iterator` produces the next value from its input, and then either yields a transformation of this value, or takes its input and yields each value from a formula based on that value. For example, the `step` function produces consecutive values but does not need to first produce an array containing all of the values.

### Writing iterative functions

**Remember:** You can't write iterative functions solely in the Puppet language.

When writing iterative functions, use the `Iterable` type instead of the more specific, individual types. The `Iterable` type has a type parameter that describes the type that is yielded in each iteration. For example, an `Array[Integer]` is also an `Iterable[Integer]`.

When writing a function that returns an `Iterator`, declare the return type as `Iterable`. This is the most flexible way to handle an `Iterator`.

For best practices on implementing iterative functions, examine existing iterative functions in Puppet and read the Ruby documentation for the helper classes these functions use. See the implementations of `each` and `map` for functions that always produce a new result, and `reverse_each` and `step` for new iterative functions that return an `Iterable` when called without a block.

For example, this is the Ruby code for the `step` function:

```
Puppet::Functions.create_function(:step) do
  dispatch :step do
    param 'Iterable', :iterable
    param 'Integer[1]', :step
  end

  dispatch :step_block do
    param 'Iterable', :iterable
    param 'Integer[1]', :step
    block_param 'Callable[1,1]', :block
  end

  def step(iterable, step)
    # produces an Iterable
    Puppet::Pops::Types::Iterable.asserted_iterable(self,
 iterable).step(step)
  end

  def step_block(iterable, step, &block)
    Puppet::Pops::Types::Iterable.asserted_iterable(self,
 iterable).step(step, &block)
    nil
  end
end
```

### Efficiently chaining iterative functions

Iterative functions are often used in chains, where the result of one function is used as the next function's parameter. A typical example is a `map`/`reduce` function, where values are first modified, and then an aggregate value is computed. For example, this use of `reverse_each` and `reduce`:

```
[1,2,3].reverse_each.reduce |$result, $x| { $result - $x }
```

The `reverse_each` function iterates over the `Array` to reverse the order of its values from `[1,2,3]` to `[3,2,1]`. The `reduce` function iterates over the `Array`, subtracting each value from the previous value. The `$result` is `0`, because 3 - 2 - 1 = 0.

Iterable types allow functions like these to execute more efficiently in a chain of calls, because they eliminate each function's need to create an intermediate copy of the mapped values in the appropriate type. In the above example, the mapped values would be the array `[3,2,1]` produced by the `reverse_each` function. The first time the `reduce` function is called, it receives the values `3` and `2` — the value `1` has not yet been computed. In the next iteration, `reduce` receives the value `1`, and the chain ends because there are no more values in the array.

### Limitations and workarounds

When you use it last in a chain, you can assign a value of `Iterator[T]` (where `T` is a data type) to a variable and pass it on. However, you cannot assign an `Iterator` to a parameter value. It's also not possible to call legacy 3.x functions with an `Iterator`.

If you assign an `Iterator` to a resource attribute, you get an error. This is because the `Iterator` type is a special algorithm-based `Iterable` that is not backed by a concrete data type. In addition, parameters in resources are serialized, and Puppet cannot serialize a temporary algorithmic result.

For example, if you used the following Puppet code:

```
notify { 'example1':
  message => [1,2,3].reverse_each,
}
```

You would recieve the following error:

```
Error while evaluating a '=>' expression, Use of an Iterator is not
 supported here
```

Puppet needs a concrete data type for serialization, but the result of `[1,2,3].reverse_each` is only a temporary `Iterator` value. To convert the `Iterator`-typed value to an `Array`, map the value.

This example results in an array by chaining the `map` function:

```
notify { 'mapped_iterator':
  message => [1,2,3].reverse_each.map |$x| { $x },
}
```

You can also use the splat operator `*` to convert the value into an array.

```
notify { 'mapped_iterator':
  message => *[1,2,3].reverse_each,
}
```

Both of these examples result in a notice containing `[3,2,1]`. If you use `*` in a context where it also unfolds, the result is the same as unfolding an array: each value of the array becomes a separate value, which results in separate arguments in a function call.

Related topics: `step` functions, `each` functions, `reduce` functions, `map` functions, `reverse_each` functions.

## Refactoring legacy 3.x functions

If you have Ruby functions written with the legacy 3.x API, refactor them to ensure that they work correctly with current versions of Puppet.

Refactoring legacy functions improves functionality and prevents errors. At minimum, refactor any extra methods in your 3.x functions, because these no longer work in Puppet.

### Extra methods

Legacy functions that contain methods defined inside the function body or outside of the function return an error, such as:

```
raise SecurityError, _("Illegal method definition of method '%{method_name}'
 on line %{line}' in legacy function") % { method_name: mname, line: mline }
```

To fix these errors, refactor your 3.x functions to the 4.x function API, where defining multiple methods is permitted.

For example, the legacy function below has been refactored into the modern API, with the following changes:

- Documentation for the function is now a comment before the call to `create_function`.
- The default dispatcher dispatches all given arguments to a method with the same name as the function.
- The `extra_method` has not been moved, but is legal in the modern API.
- Not visible in the code sample, the function has been moved from `lib/puppet/parser/functions` to `lib/puppet/functions`.

3.x API function:

```
module Puppet::Parser::Functions
  newfunction(:sample, :type => :rvalue, :doc => <<-EOS
    The function's documentation
  EOS
  ) do |arguments|
    "the returned value"
  end
```

```
  def extra_method()
  end
end
```

4.x API function:

```
# The function's documentation
Puppet::Functions.create_function(:sample) do

  def sample(*arguments)
    "the returned value"
  end

  def extra_method()
  end
end
```

#### Function call forms

Change all function calls from the form `function_***` to use the method `call_function(name, args)`.

The `function_***` form applies only to functions implemented in the 3.x API, so function with calls in that form can not call any function that has moved to the 4.x API.

For example, a 3.x function:

```
function_sprintf("%s", "example")
```

The refactored 4.x function:

```
call_function('sprintf', "%s", "example")
```

#### `:rvalue` specification

The 3.x API differentiated between functions returning a value (`:type => :rvalue`) and functions that did not return a value (`:type => :statement`). In the 4.x API, there is no such distinction. If you are refactoring a function where `:rvalue => true`, you do not need to make any changes. If you are refactoring a function where `:rvalue => false`, make sure the function returns `nil`.

#### Data values

The 4.x function API allows certain data values, such as `Regexp`, `Timespan`, and `Timestamp`. However, the 3.x API transformed these and similar data values into strings.

Review the logic in your refactored function with this in mind: instead of checking for empty strings, the function checks for `nil`. The function uses neither empty strings nor the `:undef` symbol in returned values to denote `undef`; again, use `nil` instead.

For `String`, `Integer`, `Float`, `Array`, `Hash`, and `Boolean` values, you do not need to make changes to your 3.x functions.

#### Documentation

The 4.x API supports Markdown and Puppet Strings documentation tags to document functions, including individual parameters and returned values. See the Strings documentation page for details about the correct format and style for documentation comments.

#### Namespacing

Namespace your function to the module in which it is defined, and update manifests that use it.

The function name is in the format `module_name::function_name`. For example, if the module name is `mymodule`:

```
# The function's documentation
Puppet::Functions.create_function(:'mymodule::sample') do

  def sample(*arguments)
    "the returned value"
  end

  def extra_method()
  end
end
```

The default dispatch uses the last part of the name when dispatching to a method in the function, so you only have to change the module namespace in the function's full name. You must also move the file containing the function to the correct location for 4.x API functions, `mymodule/lib/puppet/functions`.

## Secrets store integrations and agent-side data retrieval

Puppet agents can fetch or calculate data for themselves at catalog application time. One use case for this is to securely retrieve sensitive information like passwords from a secrets store.

The `Deferred` type enables these two capabilities. It instructs agents to execute a function locally to resolve a data value at the time of catalog application. When compiling catalogs, functions are normally executed on the master, with results entered into the catalog directly. The complete and fully resolved catalog is then sent to the agent for application. Starting in Puppet 6.0, you can defer the function call until the agent applies the catalog, meaning the agent calls the function on the agent instead of on the master. This way, agents can use a function to fetch data like secrets directly, rather than having the master act as an intermediary.

### Integrations with secret stores

The Forge already hosts some community modules that provide integrations with secret stores.

Modules with secret store integrations:

- Azure Key Vault: works on both the master and the server.
- Cyberark Conjur: works on the master.
- Hashicorp Vault: works on the agent.
- Cyberark AIM: works on the agent (coming soon).

### Using a `Deferred` function

An example of using the `Deferred` type to wrap a function and execute on the agent.

#### `Deferred` function example

Prior to Puppet 6.0, you used a function executed on the master to evaluate a result and store it in the catalog. See the Puppet code below, which prints the result of `myfunction`.

```
$d = myfunction("myarg1", “myarg2”)

node default {
  notify { example :
    message => $d
  }
```

```
}
```

To execute this function on the agent, wrap it with the `Deferred` type. In this case, the function name is the first parameter and the function's parameters are passed as an array. Converting the same call to `myfunction` using `Deferred` looks like this:

```
$d = Deferred(“myfunction”, ["myarg1", “myarg2”])
```

## Write a Puppet function to store secrets

Use the `Deferred` type to create a function that you add to a module to redact sensitive information.

These instructions use Puppet Development Kit (PDK), our recommended tool for creating modules. The steps are also based on RHEL 7 OS.

1. Install PDK using the the following commands:
   a) `sudo rpm -Uvh https://yum.puppet.com/puppet5-release-el-7.noarch.rpm`
   b) `sudo yum install pdk`

   You might have to restart your command-line interface for `pdk` commands to be in your path.
2. From a working directory, run the following commands. You can accept the default answers to the questions for the steps.
   a) `pdk new module mymodule`
   b) `cd mymodule`
   c) `pdk new class mymodule`
   d) `mkdir -p lib/puppet/functions`
3. Paste this code into `manifests/init.pp`.

```
# This is a simple example of calling a function at catalog apply time.
#
# @summary Demonstrates calling a Deferred function that is housed with
 this module in lib/puppet/functions/myupcase.rb
#
# @example
#   puppet apply manifests/init.pp
class mymodule {
  $d = Deferred("mymodule::myupcase", ["mysecret"])

  notify { example :
    message => $d
  }
}

class { 'mymodule': }
```

4. Paste this code into `lib/puppet/functions/myupcase.rb`

```
Puppet::Functions.create_function(:'mymodule::myupcase') do
  dispatch :up do
    param 'String', :some_string
  end

  def up(some_string)
    Puppet::Pops::Types::PSensitiveType::Sensitive.new(some_string.upcase)
  end
end
```

5. Run `/opt/puppetlabs/bin/puppet apply manifests/init.pp`. This outputs a notice.

   The use of `Sensitive` in the `up` function tells the agent not to store the cleartext value in logs or reports. On the command line and in the Puppet Enterprise console, sensitive data appears as `[redacted]`.

   **Note:** The workflow using `Deferred` functions is the same module adoption workflow that you already use for other modules; you can package functions in a module that are synced down to agents. In most cases, you add the new module to your Puppetfile.

## `Deferred` functions - notes on using

Notes for consideration when working with `Deferred` functions.

### Important info about using `Deferred`

- If an agent is applying a cached catalog, the `Deferred` function is still called at application time, and the value returned at that time is the value that is used.
- It is the responsibility of the function to handle edge cases such as providing default or cached values in cases where a remote store is unavailable.
- `Deferred` supports only the Puppet function API for Ruby.
- If a function called on the agent side does not return `Sensitive`, you can wrap the value returned by `Deferred` in a `Sensitive` type if a sensitive value is desired. For example: `$d = Sensitive(Deferred("myupcase", ["example value"]))`

# Hiera

Hiera is a built-in key-value configuration data lookup system, used for separating data from Puppet code.



## About Hiera

Puppet's strength is in reusable code. Code that serves many needs must be configurable: put site-specific information in external configuration data files, rather than in the code itself.

Puppet uses Hiera to do two things:

- Store the configuration data in key-value pairs

- Look up what data a particular module needs for a given node during catalog compilation

This is done via:

- Automatic Parameter Lookup for classes included in the catalog
- Explicit lookup calls

Hiera's hierarchical lookups follow a "defaults, with overrides" pattern, meaning you specify common data one time, and override it in situations where the default won't work. Hiera uses Puppet's facts to specify data sources, so you can structure your overrides to suit your infrastructure. While using facts for this purpose is common, data-sources can also be defined without the use of facts.

Puppet 5 comes with support for JSON, YAML, and EYAML files.

Related topics: Automatic Parameter Lookup.

## Hiera hierarchies

Hiera looks up data by following a hierarchy — an ordered list of data sources.

Hierarchies are configured in a `hiera.yaml` configuration file. Each level of the hierarchy tells Hiera how to access some kind of data source. A hierarchy is usually organized like this:

```
---
version: 5
defaults:  # Used for any hierarchy level that omits these keys.
  datadir: data         # This path is relative to hiera.yaml's directory.
  data_hash: yaml_data  # Use the built-in YAML backend.

hierarchy:
  - name: "Per-node data"                   # Human-readable name.
    path: "nodes/%{trusted.certname}.yaml"  # File path, relative to
 datadir.
                                   # ^^^ IMPORTANT: include the file
 extension!

  - name: "Per-datacenter business group data" # Uses custom facts.
    path: "location/%{facts.whereami}/%{facts.group}.yaml"

  - name: "Global business group data"
    path: "groups/%{facts.group}.yaml"

  - name: "Per-datacenter secret data (encrypted)"
    lookup_key: eyaml_lookup_key   # Uses non-default backend.
    path: "secrets/%{facts.whereami}.eyaml"
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
      pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem

  - name: "Per-OS defaults"
    path: "os/%{facts.os.family}.yaml"

  - name: "Common data"
    path: "common.yaml"
```

In this example, every level configures the path to a YAML file on disk.

### Hierarchies interpolate variables

Most levels of a hierarchy interpolate variables into their configuration:

```
path: "os/%{facts.os.family}.yaml"
```

The percent-and-braces `%{variable}` syntax is a Hiera interpolation token. It is similar to the Puppet language's `${expression}` interpolation tokens. Wherever you use an interpolation token, Hiera determines the variable's value and inserts it into the hierarchy.

The `facts.os.family` uses the Hiera special `key.subkey` notation for accessing elements of hashes and arrays. It is equivalent to `$facts['os']['family']` in the Puppet language but the 'dot' notation produces an empty string instead of raising an error if parts of the data is missing. Make sure that an empty interpolation does not end up matching an unintended path.

You can only interpolate values into certain parts of the config file. For more info, see the `hiera.yaml` format reference.

With node-specific variables, each node gets a customized set of paths to data. The hierarchy is always the same.

### Hiera searches the hierarchy in order

After Hiera replaces the variables to make a list of concrete data sources, it checks those data sources in the order they were written.

Generally, if a data source doesn't exist, or doesn't specify a value for the current key, Hiera skips it and moves on to the next source, until it finds one that exists — then it uses it. Note that this is the default merge strategy, but does not always apply, for example, Hiera can use data from all data sources and merge the result.

Earlier data sources have priority over later ones. In the example above, the node-specific data has the highest priority, and can override data from any other level. Business group data is separated into local and global sources, with the local one overriding the global one. Common data used by all nodes always goes last.

That's how Hiera's "defaults, with overrides" approach to data works — you specify common data at lower levels of the hierarchy, and override it at higher levels for groups of nodes with special needs.

### Layered hierarchies

Hiera uses layers of data with a `hiera.yaml` for each layer.

Each layer can configure its own independent hierarchy. Before a lookup, Hiera combines them into a single super-hierarchy: global → environment → module.

**Note:** There is a fourth layer - `default_hierarchy` - that can be used in a module's `hiera.yaml`. It only comes into effect when there is no data for a key in any of the other regular hierarchies

Assume the example above is an environment hierarchy (in the production environment). If we also had the following global hierarchy:

```
---
version: 5
hierarchy:
  - name: "Data exported from our old self-service config tool"
    path: "selfserve/%{trusted.certname}.json"
    data_hash: json_data
    datadir: data
```

And the NTP module had the following hierarchy for default data:

```
---
version: 5
hierarchy:
  - name: "OS values"
    path: "os/%{facts.os.name}.yaml"
  - name: "Common values"
    path: "common.yaml"
defaults:
  data_hash: yaml_data
  datadir: data
```

Then in a lookup for the `ntp::servers` key, `thrush.example.com` would use the following combined hierarchy:

- `<CODEDIR>/data/selfserve/thrush.example.com.json`
- `<CODEDIR>/environments/production/data/nodes/thrush.example.com.yaml`
- `<CODEDIR>/environments/production/data/location/belfast/ops.yaml`
- `<CODEDIR>/environments/production/data/groups/ops.yaml`
- `<CODEDIR>/environments/production/data/os/Debian.yaml`
- `<CODEDIR>/environments/production/data/common.yaml`
- `<CODEDIR>/environments/production/modules/ntp/data/os/Ubuntu.yaml`
- `<CODEDIR>/environments/production/modules/ntp/data/common.yaml`

The combined hierarchy works the same way as a layer hierarchy. Hiera skips empty data sources, and either returns the first found value or merges all found values.

**Note:** By default, `datadir` refers to the directory named 'data' next to the `hiera.yaml`.

### Tips for making a good hierarchy

- Make a short hierarchy. Data files are easier to work with.
- Use the roles and profiles method to manage less data in Hiera. Sorting hundreds of class parameters is easier than sorting thousands.
- If the built-in facts don't provide an easy way to represent differences in your infrastructure, make custom facts. For example, create a custom datacenter fact that is based on information particular to your network layout so that each datacenter is uniquely identifiable.
- Give each environment – production, test, development – its own hierarchy.

Related topics: codedir, confdir.

## Hiera configuration layers

Hiera uses three independent layers of configuration. Each layer has its own hierarchy, and they're linked into one super-hierarchy before doing a lookup.

The three layers are searched in the following order: global → environment → module. Hiera searches every data source in the global layer's hierarchy before checking any source in the environment layer.

### The global layer

The configuration file for the global layer is located, by default, in `$confdir/hiera.yaml`. You can change the location by changing the `hiera_config` setting in `puppet.conf`.

Hiera has one global hierarchy. Because it goes before the environment layer, it's useful for temporary overrides, for example, when your ops team needs to bypass its normal change processes.

The global layer is the only place where legacy Hiera 3 backends can be used - it's an important piece of the transition period when you migrate you backends to support Hiera 5. It supports the following config formats: `hiera.yaml` v5, `hiera.yaml` v3 (deprecated).

Other than the above use cases, try to avoid the global layer. All normal data should be specified in the environment layer.

### The environment layer

The configuration file for the global layer is located, by default, in `<ENVIRONMENT DIR>/hiera.yaml`.

The environment layer is where most of your Hiera data hierarchy definition happens. Every Puppet environment has its own hierarchy configuration, which applies to nodes in that environment. Supported config formats include: v5, v3 (deprecated).

#### The module layer

The configuration file for a module layer is located, by default, in a module's `<MODULE>/hiera.yaml`.

The module layer sets default values and merge behavior for a module's class parameters. It is a convenient alternative to the `params.pp` pattern.

**Note:** To get the exact same behaviour as `params.pp`, the `default_hierarchy` should be used, as those bindings are excluded from merges. When placed in the regular hierarchy in the module's hierarchy the bindings are merged when a merge lookup is performed.

It comes last in Hiera's lookup order, so environment data set by a user overrides the default data set by the module's author.

Every module can have its own hierarchy configuration. You can only bind data for keys in the module's namespace. For example:

| Lookup key | Relevant module hierarchy |
|---|---|
| `ntp::servers` | `ntp` |
| `jenkins::port` | `jenkins` |
| `secure_server` | `(none)` |

Hiera uses the `ntp` module's hierarchy when looking up `ntp::servers`, but uses the `jenkins` module's hierarchy when looking up `jenkins::port`. Hiera never checks the module for a key beginning with `jenkins::`.

When you use the lookup function for keys that don't have a namespace (for example, `secure_server`), the module layer is not consulted.

The three-layer system means that each environment has its own hierarchy, and so do modules. You can make hierarchy changes on an environment-by-environment basis. Module data is also customizable.

## Getting started with Hiera

This page introduces the basic concepts and tasks to get you started with Hiera, including how to create a hiera.yaml config file and write data. It is the foundation for understanding the more advanced topics described in the rest of the Hiera documentation.

**Related information**

Hiera configuration layers on page 398
Hiera uses three independent layers of configuration. Each layer has its own hierarchy, and they're linked into one super-hierarchy before doing a lookup.

Merge behaviors on page 410
There are four merge behaviors to choose from: `first`, `unique`, `hash`, and `deep`.

### Create a `hiera.yaml` config file

The Hiera config file is called `hiera.yaml`. Each environment should have its own `hiera.yaml` file.

In the main directory of one of your environments, create a new file called `hiera.yaml`. Paste the following contents into it:

```
# <ENVIRONMENT>/hiera.yaml
---
version: 5

hierarchy:
  - name: "Per-node data"                   # Human-readable name.
    path: "nodes/%{trusted.certname}.yaml"  # File path, relative to
 datadir.
```

```
                                   # ^^^ IMPORTANT: include the file
extension!

  - name: "Per-OS defaults"
    path: "os/%{facts.os.family}.yaml"

  - name: "Common data"
    path: "common.yaml"
```

This file is in a format called YAML, which is used extensively throughout Hiera.

For more information on YAML, see YAML Cookbook.

**Related information**

Config file syntax on page 403
The `hiera.yaml` file is a YAML file, containing a hash with up to four top-level keys.

## The hierarchy

The `hiera.yaml` file configures a hierarchy: an ordered list of data sources.

Hiera searches these data sources in the order they are written. Higher-priority sources override lower-priority ones. Most hierarchy levels use variables to locate a data source, so that different nodes get different data.

This is the core concept of Hiera: a defaults-with-overrides pattern for data lookup, using a node-specific list of data sources.

**Related information**

Interpolation on page 415
In Hiera you can insert, or interpolate, the value of a variable into a string, using the syntax `%{variable}`.

Hiera hierarchies on page 396
Hiera looks up data by following a hierarchy — an ordered list of data sources.

## Write data: Create a test class

A test class writes the data it receives to a temporary file — on the agent when applying the catalog.

Hiera is used with Puppet code, so the first step is to create a Puppet class for testing.

1. If you do not already use the roles and profiles method, create a module named `profile`. Profiles are wrapper classes that use multiple component modules to configure a layered technology stack. See the related topic links for more information.
2. Use Puppet Development Kit ( PDK) to create a class called hiera_test.pp in your `profile` module.
3. Add the following code you your `hiera_test.pp` file:

```
# /etc/puppetlabs/code/environments/production/modules/profile/manifests/
hiera_test.pp
class profile::hiera_test (
  Boolean             $ssl,
  Boolean             $backups_enabled,
  Optional[String[1]] $site_alias = undef,
) {
  file { '/tmp/hiera_test.txt':
    ensure  => file,
    content => @("END"),
               Data from profile::hiera_test
               -----
               profile::hiera_test::ssl: ${ssl}
               profile::hiera_test::backups_enabled: ${backups_enabled}
               profile::hiera_test::site_alias: ${site_alias}
               |END
    owner   => root,
    mode    => '0644',
```

```
    }
}
```

The test class uses class parameters to request configuration data. Puppet looks up class parameters in Hiera, using `<CLASS NAME>::<PARAMETER NAME>` as the lookup key.

4. Make a manifest that includes the class:

```
# site.pp
include profile::hiera_test
```

5. Compile the catalog and observe that this fails because there are required values.
6. To provide values for the missing class parameters, set the following keys in your Hiera data:

| Parameter | Hiera key |
|---|---|
| `$ssl` | `profile::hiera_test::ssl` |
| `$backups_enabled` | `profile::hiera_test::backups_enabled` |
| `$site_alias` | `profile::hiera_test::site_alias` |

7. Compile again and observe that the parameters are now automatically looked up.

### Related information

The Puppet lookup function on page 419
The `lookup` function uses Hiera to retrieve a value for a given key.

## Write data: Set values in common data

Set values in your common data — the level at the bottom of your hierarchy.

This hierarchy level uses the YAML backend for data, which means the data goes into a YAML file. To know where to put that file, combine the following pieces of information:

- The current environment's directory.
- The data directory, which is a subdirectory of the environment. By default, it's `<ENVIRONMENT>/data`.
- The file path specified by the hierarchy level.

In this case, `/etc/puppetlabs/code/environments/production/` + `data/` + `common.yaml`.

Open that YAML file in an editor, and set values for two of the class's parameters.

```
# /etc/puppetlabs/code/environments/production/data/common.yaml
---
profile::hiera_test::ssl: false
profile::hiera_test::backups_enabled: true
```

The third parameter, `$site_alias`, has a default value defined in code, so you can omit it from the data.

## Write data: Set per-operating system data

The second level of the hierarchy uses the `os` fact to locate its data file. This means it can use different data files depending on the operating system of the current node.

For this example, suppose that your developers use MacBook laptops, which have an OS family of `Darwin`. If a developer is running an app instance on their laptop, it should not send data to your production backup server, so set `$backups_enabled` to `false`.

If you do not run Puppet on any Mac laptops, choose an OS family that is meaningful to your infrastructure.

1. Locate the data file, by replacing `%{facts.os.family}` with the value you are targeting:

   `/etc/puppetlabs/code/environments/production/data/` + `os/` + `Darwin` + `.yaml`

2. Add the following contents:

```
# /etc/puppetlabs/code/environments/production/data/os/Darwin.yaml
---
profile::hiera_test::backups_enabled: false
```

3. Compile to observe that the override takes effect.

Related topics: the os fact.

## Write data: Set per-node data

The highest level of the example hierarchy uses the value of `$trusted['certname']` to locate its data file, so you can set data by name for each individual node.

This example supposes you have a server named `jenkins-prod-03.example.com`, and configures it to use SSL and to serve this application at the hostname `ci.example.com`. To try this out, choose the name of a real server that you can run Puppet on.

1. To locate the data file, replace `%{trusted.certname}`with the node name you're targeting:

   `/etc/puppetlabs/code/environments/production/data/` + `nodes/` + `jenkins-prod-03.example.com` + `.yaml`
2. Open that file in an editor and add the following contents:

```
# /etc/puppetlabs/code/environments/production/data/nodes/jenkins-
prod-03.example.com.yaml
---
profile::hiera_test::ssl: true
profile::hiera_test::site_alias: ci.example.com
```

3. Compile to observe that the override takes effect.

Related topics: $trusted['certname'].

## Testing Hiera data on the command line

As you set Hiera data or rearrange your hierarchy, it is important to double-check the data a node receives.

The `puppet lookup` command helps test data interactively. For example:

```
puppet lookup profile::hiera_test::backups_enabled --environment production
 --node jenkins-prod-03.example.com
```

This returns the value `true`.

To use the `puppet lookup` command effectively:

- Run the command on a Puppet Server node, or on another node that has access to a full copy of your Puppet code and configuration.
- The node you are testing against should have contacted the server at least one time as this makes the facts for that node available to the `lookup` command (otherwise you need to supply the facts yourself on the command line).
- Make sure the command uses the global `confdir` and `codedir`, so it has access to your live data. If you're not running `puppet lookup` as root user, specify `--codedir` and `--confdir` on the command line.
- If you use PuppetDB, you can use any node's facts in a lookup by specifying `--node <NAME>`. Hiera can automatically get that node's real facts and use them to resolve data.
- If you do not use PuppetDB, or if you want to test for a set of facts that don't exist, provide facts in a YAML or JSON file and specify that file as part of the command with `--facts <FILE>`. To get a file full of facts, rather than creating one from scratch, run `facter -p --json > facts.json` on a node that is similar to the node you want to examine, copy the `facts.json` file to your Puppet Server node, and edit it as needed.
  - Puppet Development Kit comes with predefined fact sets for a variety of platforms. You can use those if you want to test against platforms you do not have, or if you want "typical facts" for a kind of platform.

- If you are not getting the values you expect, try re-running the command with `--explain`. The `--explain` flag makes Hiera output a full explanation of which data sources it searched and what it found in them.

Related topics: The puppet lookup command, confdir, codedir.

## Configuring Hiera

The Hiera configuration file is called `hiera.yaml`. It configures the hierarchy for a given layer of data.

**Related information**

Hiera configuration layers on page 398
Hiera uses three independent layers of configuration. Each layer has its own hierarchy, and they're linked into one super-hierarchy before doing a lookup.

Hiera hierarchies on page 396
Hiera looks up data by following a hierarchy — an ordered list of data sources.

### Location of `hiera.yaml` files

There are several `hiera.yaml` files in a typical deployment. Hiera uses three layers of configuration, and the module and environment layers typically have multiple instances.

The configuration file locations for each layer:

| Layer | Location | Example |
|---|---|---|
| Global | `$confdir/hiera.yaml` | `/etc/puppetlabs/puppet/hiera.yaml` `C:\ProgramData\PuppetLabs\puppet\etc\hiera.yaml` |
| Environment | `<ENVIRONMENT>/hiera.yaml` | `/etc/puppetlabs/code/environments/production/hiera.yaml` `C:\ProgramData\PuppetLabs\code\environments\production\hiera.yaml` |
| Module | `<MODULE>/hiera.yaml` | `/etc/puppetlabs/code/environments/production/modules/ntp/hiera.yaml` `C:\ProgramData\PuppetLabs\code\environments\production\modules\ntp\hiera.yaml` |

**Note:** To change the location for the global layer's `hiera.yaml` set the `hiera_config` setting in your `puppet.conf` file.

Hiera searches for data in the following order: global → environment → module. For more information, see Hiera configuration layers.

Related topics: codedir, Environments, Modules fundamentals.

### Config file syntax

The `hiera.yaml` file is a YAML file, containing a hash with up to four top-level keys.

The following keys are in a `hiera.yaml` file:

- `version` - Required. Must be the number 5, with no quotes.
- `defaults` - A hash, which can set a default `datadir`, `backend`, and `options` for hierarchy levels.

- `hierarchy` - An array of hashes, which configures the levels of the hierarchy.
- `default_hierarchy` - An array of hashes, which sets a default hierarchy to be used only if the normal hierarchy entries do not result in a value. Only allowed in a module's `hiera.yaml.`

```
version: 5
defaults:  # Used for any hierarchy level that omits these keys.
  datadir: data         # This path is relative to hiera.yaml's directory.
  data_hash: yaml_data  # Use the built-in YAML backend.

hierarchy:
  - name: "Per-node data"                   # Human-readable name.
    path: "nodes/%{trusted.certname}.yaml"  # File path, relative to
 datadir.
                                   # ^^^ IMPORTANT: include the file
 extension!

  - name: "Per-datacenter business group data" # Uses custom facts.
    path: "location/%{facts.whereami}/%{facts.group}.yaml"

  - name: "Global business group data"
    path: "groups/%{facts.group}.yaml"

  - name: "Per-datacenter secret data (encrypted)"
    lookup_key: eyaml_lookup_key   # Uses non-default backend.
    path: "secrets/%{facts.whereami}.eyaml"
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
      pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem

  - name: "Per-OS defaults"
    path: "os/%{facts.os.family}.yaml"

  - name: "Common data"
    path: "common.yaml"
```

**Note:** When writing in Hiera YAML files, do not use hard tabs for indentation.

### The default configuration

If you omit the `hierarchy` or `defaults` keys, Hiera uses the following default values.

```
version: 5
hierarchy:
  - name: Common
    path: common.yaml
defaults:
  data_hash: yaml_data
  datadir: data
```

These defaults are only used if the file is present and specifies `version: 5`. If `hiera.yaml` is absent, it disables Hiera for that layer. If it specifies a different version, different defaults apply.

### The defaults key

The `defaults` key sets default values for the `lookup function` and `datadir` keys, which lets you omit those keys in your hierarchy levels. The value of defaults must be a hash, which can have up to three keys: `datadir`, `options`, and one of the mutually exclusive `lookup function` keys.

datadir: a default value for `datadir`, used for any file-based hierarchy level that doesn't specify its own. If not given, the `datadir` is the directory `data` in the same directory as the `hiera.yaml` configuration file.

options: a default value for `options`, used for any hierarchy level that does not specify its own.

The lookup function keys: used for any hierarchy level that doesn't specify its own. This must be one of:

- `data_hash` - produces a hash of key-value pairs (typically from a data file)
- `lookup_key` - produces values key by key (typically for a custom data provider)
- `data_dig` - produces values key by key (for a more advanced data provider)
- `hiera3_backend` - a data provider that calls out to a legacy Hiera 3 backend (global layer only).

For the built-in data providers — YAML, JSON, and HOCON — the key is always `data_hash` and the value is one of `yaml_data`, `json_data`, or `hocon_data`. To set a custom data provider as the default, see the data provider documentation. Whichever key you use, the value must be the name of the custom Puppet function that implements the lookup function.

#### The hierarchy key

The `hierarchy` key configures the levels of the hierarchy. The value of `hierarchy` must be an array of hashes.

Indent the hash's keys by four spaces, so they line up with the first key. Put an empty line between hashes, to visually distinguish them. For example:

```
hierarchy:
  - name: "Per-node data"
    path: "nodes/%{trusted.certname}.yaml"

  - name: "Per-datacenter business group data"
    path: "location/%{facts.whereami}/%{facts.group}.yaml"
```

#### The `default_hierarchy` key

The `default_hierarchy` key is a top-level key. It is initiated when, and only when, the lookup in the regular hierarchy does not find a value. Within this default hierarchy, the normal merging rules apply. The `default_hierarchy` is not permitted in environment or global layers.

If `lookup_options` is used, the values found in the regular hierarchy have no effect on the values found in the `default_hierarchy`, and vice versa. A merge parameter, given in a call to lookup, is only used in the regular hierarchy. It does not affect how a value in the default hierarchy is assembled. The only way to influence that, is to use `lookup_options`, found in the default hierarchy.

For more information about the YAML file, see YAML.

#### Related information

Hiera hierarchies on page 396
Hiera looks up data by following a hierarchy — an ordered list of data sources.

## Configuring a hierarchy level: built-in backends

Hiera has three built-in backends: YAML, JSON, and HOCON. All of these use files as data sources.

You can use any combination of these backends in a hierarchy, and can also combine them with custom backends. But if most of your data is in one file format, set default values for the `datadir` and `data_hash` keys.

Each YAML/JSON/HOCON hierarchy level needs the following keys:

- `name` — A name for this level, shown in debug messages and `--explain` output.
- `path`, `paths`, `glob`, `globs`, or `mapped_paths` (choose one) — The data files to use for this hierarchy level.
  - These paths are relative to the `datadir`, they support variable interpolation, and they require a file extension. See “Specifying file paths” for more details.

- `data_hash` — Which backend to use. Can be omitted if you set a default. The value must be one of the following:
  - `yaml_data` for YAML.
  - `json_data` for JSON.
  - `hocon_data` for HOCON.
- `datadir` — The directory where data files are kept. Can be omitted if you set a default.
  - This path is relative to `hiera.yaml`'s directory: if the config file is at `/etc/puppetlabs/code/environments/production/hiera.yaml` and the `datadir` is set to data, the full path to the data directory is `/etc/puppetlabs/code/environments/production/data`.
  - In the global layer, you can optionally set the `datadir` to an absolute path; in the other layers, it must always be relative.

For more information on built-in backends, see YAML, JSON, HOCON.

### Related information

Interpolate a Puppet variable on page 416
The most common thing to interpolate is the value of a Puppet top scope variable.

### Specifying file paths

Options for specifying a file path.

| Key | Data type | Expected value |
|---|---|---|
| `path` | String | One file path. |
| `paths` | Array | Any number of file paths. This acts like a sub-hierarchy: if multiple files exist, Hiera searches all of them, in the order in which they're written. |
| `glob` | String | One shell-like glob pattern, which might match any number of files. If multiple files are found, Hiera searches all of them in alphanumerical order. |
| `globs` | Array | Any number of shell-like glob patterns. If multiple files are found, Hiera searches all of them in alphanumerical order (ignoring the order of the globs). |
| `mapped_paths` | Array or Hash | A fact that is a collection (array or hash) of values. Hiera expands these values to produce an array of paths. |

**Note:** You can only use one of these keys in a given hierarchy level.

Explicit file extensions are required, for example, `common.yaml`, not `common`.

File paths are relative to the `datadir`: if the full `datadir` is `/etc/puppetlabs/code/environments/production/data` and the file path is set to `"nodes/%{trusted.certname}.yaml"`, the full path to the file is `/etc/puppetlabs/code/environments/production/data/nodes/<NODE NAME>.yaml`.

**Note:** Hierarchy levels should interpolate variables into the path.

Globs are implemented with Ruby's `Dir.glob` method:

- One asterisk (`*`) matches a run of characters.
- Two asterisks (`**`) matches any depth of nested directories.

- A question mark (`?`) matches one character.
- Comma-separated lists in curly braces (`{one,two}`) match any option in the list.
- Sets of characters in square brackets (`[abcd]`) match any character in the set.
- A backslash (`\`) escapes special characters.

Example:

```
- name: "Domain or network segment"
    glob: "network/**/{%{facts.networking.domain},
%{facts.networking.interfaces.en0.bindings.0.network}}.yaml"
```

The `mapped_paths` key must contain three string elements, in the following order:

- A scope variable that points to a collection of strings.
- The variable name that is mapped to each element of the collection.
- A template where that variable can be used in interpolation expressions.

For example, a fact named `$services` contains the array `["a", "b", "c"]`. The following configuration has the same results as if paths had been specified to be `[service/a/common.yaml, service/b/common.yaml, service/c/common.yaml]`.

```
- name: Example
    mapped_paths: [services, tmp, "service/%{tmp}/common.yaml"]
```

**Related information**

Interpolation on page 415
In Hiera you can insert, or interpolate, the value of a variable into a string, using the syntax `%{variable}`.

The hierarchy on page 400
The `hiera.yaml` file configures a hierarchy: an ordered list of data sources.

## Configuring a hierarchy level: `hiera-eyaml`

Hiera 5 ( Puppet 4.9.3 and later) includes a native interface for the Hiera eyaml extension, which keeps data encrypted on disk but lets Puppet read it during catalog compilation.

To learn how to create keys and edit encrypted files, see the Hiera eyaml documentation.

Within `hiera.yaml`, the eyaml backend resembles the standard built-in backends, with a few differences: it uses `lookup_key` instead of `data_hash`, and requires an `options` key to locate decryption keys. Note that the eyaml backend can read regular yaml files as well as yaml files with encrypted data.

**Important:** To use the eyaml backend, you must have the `hiera-eyaml` gem installed where Puppet can use it. It's included in Puppet Server since version 5.2.0, so you just need to make it available for command line usage. To enable eyaml on the command line and with `puppet apply`, use `sudo /opt/puppetlabs/puppet/bin/gem install hiera-eyaml`.

Each eyaml hierarchy level needs the following keys:

- `name` — A name for this level, shown in debug messages and `--explain` output.
- `lookup_key` — Which backend to use. The value must be `eyaml_lookup_key`. Use this instead of the `data_hash` setting.
- `path`, `paths`, `mapped_paths`, `glob`, or `globs` (choose one) — The data files to use for this hierarchy level. These paths are relative to the datadir, they support variable interpolation, and they require a file extension. In this case, you'll usually use `.eyaml`. They work the same way they do for the standard backends.
- `datadir` — The directory where data files are kept. Can be omitted if you set a default. Works the same way it does for the standard backends.

- `options` — A hash of options specific to `hiera-eyaml`, mostly used to configure decryption. For the default encryption method, this hash must have the following keys:
  - `pkcs7_private_key` — The location of the PKCS7 private key to use.
  - `pkcs7_public_key` — The location of the PKCS7 public key to use.
  - If you use an alternate encryption plugin, its docs should specify which options to set. Set an `encrypt_method` option, plus some plugin-specific options to replace the `pkcs7` ones.
  - You can use normal strings as keys in this hash; you don't need to use symbols.

The file path key and the options key both support variable interpolation.

An example hierarchy level:

```
hierarchy:
  - name: "Per-datacenter secret data (encrypted)"
    lookup_key: eyaml_lookup_key
    path: "secrets/%{facts.whereami}.eyaml"
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
      pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem
```

**Related information**

Interpolation on page 415
In Hiera you can insert, or interpolate, the value of a variable into a string, using the syntax `%{variable}`.

## Configuring a hierarchy level: legacy Hiera 3 backends

If you rely on custom data backends designed for Hiera 3, you can use them in your global hierarchy. They are not supported at the environment or module layers.

**Note:** This feature is a temporary measure to let you start using new features while waiting for backend updates.

Each legacy hierarchy level needs the following keys:

- `name` — A name for this level, shown in debug messages and `--explain` output.
- `path` or `paths` (choose one) — The data files to use for this hierarchy level.
  - For file-based backends, include the file extension, even though you would have omitted it in the v3 `hiera.yaml` file.
  - For non-file backends, don't use a file extension.
- `hiera3_backend` — The legacy backend to use. This is the same name you'd use in the v3 config file's `:backends` key.
- `datadir` — The directory where data files are kept. Set this only if your backend required a `:datadir` setting in its backend-specific options.
  - This path is relative to `hiera.yaml`'s directory: if the config file is at `/etc/puppetlabs/code/environments/production/hiera.yaml` and the datadir is set to `data`, the full path to the data directory is `/etc/puppetlabs/code/environments/production/data`. Note that Hiera v3 uses 'hieradata' instead of 'data'.
  - In the global layer, you can optionally set the `datadir` to an absolute path.
- `options` — A hash, with any backend-specific options (other than `datadir`) required by your backend. In the v3 config, this would have been in a top-level key named after the backend. You can use normal strings as keys. Hiera converts them to symbols for the backend.

The following example shows roughly equivalent v3 and v5 `hiera.yaml` files using legacy backends:

```
# hiera.yaml v3
---
:backends:
  - mongodb
  - xml
```

```
:mongodb:
  :connections:
    :dbname: hdata
    :collection: config
    :host: localhost

:xml:
  :datadir: /some/other/dir

:hierarchy:
  - "%{trusted.certname}"
  - "common"


# hiera.yaml v5
---
version: 5
hierarchy:
  - name: MongoDB
    hiera3_backend: mongodb
    paths:
      - "%{trusted.certname}"
      - common
    options:
      connections:
        dbname: hdata
        collection: config
        host: localhost

  - name: Data in XML
    hiera3_backend: xml
    datadir: /some/other/dir
    paths:
      - "%{trusted.certname}.xml"
      - common.xml
```

## Configuring a hierarchy level: general format

Hiera supports custom backends.

Each hierarchy level is represented by a hash which needs the following keys:

- `name` — A name for this level, shown in debug messages and `--explain` output.
- A backend key, which must be one of:
    - `data_hash`
    - `lookup_key`
    - `data_dig` — a more specialized form of `lookup_key`, suitable when the backend is for a database. `data_dig` resolves dot separated keys, whereas `lookup_key` does not.
    - `hiera3_backend` (global layer only)

- A path or URI key — only if required by the backend. These keys support variable interpolation. The following path/URI keys are available:
  - `path`
  - `paths`
  - `mapped_paths`
  - `glob`
  - `globs`
  - `uri`
  - `uris` - these paths or URIs work the same way they do for the built-in backends. Hiera handles the work of locating files, so any backend that supports `path` automatically supports `paths`, `glob`, and `globs`. `uri` (string) and `uris` (array) can represent any kind of data source. Hiera does not ensure URIs are resolvable before calling the backend, and does not need to understand any given URI schema. A backend can omit the path/URI key, and rely wholly on the `options` key to locate its data.
- `datadir` — The directory where data files are kept: the path is relative to hiera.yaml's directory. Only required if the backend uses the `path(s)` and `glob(s)` keys, and can be omitted if you set a default.
- `options` — A hash of extra options for the backend; for example, database credentials or the location of a decryption key. All values in the `options` hash support variable interpolation.

Whichever key you use, the value must be the name of a function that implements the backend API. Note that the choice here is made by the implementer of the particular backend, not the user.

For more information, see custom Puppet function.

**Related information**

Custom backends overview on page 423
A backend is a custom Puppet function that accepts a particular set of arguments and whose return value obeys a particular format. The function can do whatever is necessary to locate its data.

Interpolation on page 415
In Hiera you can insert, or interpolate, the value of a variable into a string, using the syntax `%{variable}`.

# Creating and editing data

Important aspects of using Hiera are merge behavior and interpolation.

## Set the merge behavior for a lookup

When you look up a key in Hiera, it is common for multiple data sources to have different values for it. By default, Hiera returns the first value it finds, but it can also continue searching and merge all the found values together.

1. You can set the merge behavior for a lookup in two ways:
   - At lookup time. This works with the `lookup` function, but does not support automatic class parameter lookup.
   - In Hiera data, with the `lookup_options` key. This works for both manual and automatic lookups. It also lets module authors set default behavior that users can override.
2. With both of these methods, specify a merge behavior as either a string, for example, `'first'` or a hash, for example `{'strategy' => 'first'}`. The hash syntax is useful for `deep` merges (where extra options are available), but it also works with the other merge types.

**Related information**

The Puppet lookup function on page 419
The `lookup` function uses Hiera to retrieve a value for a given key.

## Merge behaviors

There are four merge behaviors to choose from: `first`, `unique`, `hash`, and `deep`.

When specifying a merge behavior, use one of the following identifiers:

- `'first'`, `{'strategy' => 'first'}`, or nothing.
- `'unique'` or `{'strategy' => 'unique'}`.
- `'hash'` or `{'strategy' => 'hash'}`.
- `'deep'` or `{'strategy' => 'deep', <OPTION> => <VALUE>, ...}`. Valid options:
  - `'knockout_prefix'` - string or `undef`; default is `undef`
  - `'sort_merged_arrays'` - Boolean; default is `false`
  - `'merge_hash_arrays'` - Boolean; default is `false`

### First

A first-found lookup doesn't merge anything: it returns the first value found, and ignores the rest. This is Hiera's default behavior.

Specify this merge behavior with one of these:

- `'first'`
- `{'strategy' => 'first'}`
- Nothing (because it's the default)

### Unique

A unique merge (also called an array merge) combines any number of array and scalar (string, number, boolean) values to return a merged, flattened array with all duplicate values removed. The lookup fails if any of the values are hashes. The result is ordered from highest priority to lowest.

Specify this merge behavior with one of these:

- `'unique'`
- `{'strategy' => 'unique'}`

### Hash

A hash merge combines the keys and values of any number of hashes to return a merged hash. The lookup fails if any of the values aren't hashes.

If multiple source hashes have a given key, Hiera uses the value from the highest priority data source: it won't recursively merge the values.

Hashes in Puppet preserve the order in which their keys are written. When merging hashes, Hiera starts with the lowest priority data source. For each higher priority source, it appends new keys at the end of the hash and updates existing keys in place.

```
# web01.example.com.yaml
mykey:
  d: "per-node value"
  b: "per-node override"
# common.yaml
mykey:
  a: "common value"
  b: "default value"
  c: "other common value"


`lookup('mykey', {merge => 'hash'})
```

Returns the following:

```
{
  a => "common value",
```

```
  b => "per-node override", # Using value from the higher-priority source,
 but
                            # preserving the order of the lower-priority
 source.
  c => "other common value",
  d => "per-node value",
}
```

Specify this merge behavior with one of these:

- `'hash'`
- `{'strategy' => 'hash'}`

#### Deep

A deep merge combines the keys and values of any number of hashes to return a merged hash.

If the same key exists in multiple source hashes, Hiera recursively merges them:

- Hash values are merged with another deep merge.
- Array values are merged. This differs from the unique merge. The result is ordered from lowest priority to highest, which is the reverse of the unique merge's ordering. The result is not flattened, so it can contain nested arrays. The `merge_hash_arrays` and `sort_merged_arrays` options can make further changes to the result.
- Scalar (String, Number, Boolean) values use the highest priority value, like in a first-found lookup.

Specify this merge behavior with one of these:

- `'deep'`
- `{'strategy' => 'deep', <OPTION> => <VALUE>, ...}` — Adjust the merge behavior with these additional options:
  - `'knockout_prefix'` (String or `undef`) - A string prefix to indicate a value should be removed from the final result. Defaults to `undef`, which turns off this option.
  - `'sort_merged_arrays'` (Boolean) - Whether to sort all arrays that are merged together. Defaults to `false`.
  - `'merge_hash_arrays'` (Boolean) - Whether to deep-merge hashes within arrays, by position. For example, `[ {a => high}, {b => high} ]` and `[ {c => low}, {d => low} ]` would be merged as `[ {c => low, a => high}, {d => low, b => high} ]`. Defaults to `false`.

**Note:** Unlike a hash merge, a deep merge can also accept arrays as the root values. It merges them with its normal array merging behavior, which differs from a unique merge as described above. This does not apply to the deprecated Hiera 3 `hiera_hash` function, which can be configured to do deep merges but can't accept arrays.

### Set merge behavior at lookup time

Use merge behaviour at lookup time to override preconfigured merge behavior for a key.

Use the `lookup` function or the `puppet lookup` command to provide a merge behavior as an argument or flag.

Function example:

```
# Merge several arrays of class names into one array:
lookup('classes', {merge => 'unique'})
```

Command line example:

```
$ puppet lookup classes --merge unique --environment production --explain
```

**Note:** Each of the deprecated `hiera_*` functions is locked to one particular merge behavior. (For example, Hiera only merges first-found, and `hiera_array` only performs a unique merge.)

## Set lookup_options to refine the result of a lookup

You can set lookup_options to further refine the result of a lookup, including defining merge behavior and using the convert_to key to get automatic type conversion.

### The lookup_options format

The value of lookup_options is a hash. It follows this format:

```
lookup_options:
 <NAME or REGEXP>:
   merge: <MERGE BEHAVIOR>
```

Each key is either the full name of a lookup key (like ntp::servers) or a regular expression (like '^profile::(.*)::users$'). In a module's data, you can configure lookup keys only within that module's namespace: the ntp module can set options for ntp::servers, but the apache module can't.

Each value is a hash with either a merge key, a convert_to key, or both. A merge behavior can be a string or a hash, and the type for type conversion is either a Puppet type, or an array with a type and additional arguments.

lookup_options is a reserved key in Hiera. You can't put other kinds of data in it, and you can't look it up directly.

### Location for setting lookup_options

You can set lookup_options metadata keys in Hiera data sources, including module data, which controls the default merge behavior for other keys in your data. Hiera uses a key's configured merge behavior in any lookup that doesn't explicitly override it.

**Note:** Set lookup_options in the data sources of your backend; **don't put it in the hiera.yaml file**. For example, you can set lookup_options in common.yaml.

### Defining Merge Behavior with lookup_options

In your Hiera data source, set the lookup_options key to configure merge behavior:

```
# <ENVIRONMENT>/data/common.yaml
lookup_options:
  ntp::servers:     # Name of key
    merge: unique   # Merge behavior as a string
  "^profile::(.*)::users$": # Regexp: `$users` parameter of any profile
 class
    merge:          # Merge behavior as a hash
      strategy: deep
      merge_hash_arrays: true
```

Hiera uses the configured merge behaviors for these keys.

**Note:** The lookup_options settings have no effect if you are using the deprecated hiera_* functions, which define for themselves how they do the lookup. To take advantage of lookup_options, use the lookup function or Automatic Parameter Lookup (APL).

### Overriding merge behavior

When Hiera is given lookup options, a hash merge is performed. Higher priority sources override lower priority lookup options for individual keys. You can configure a default merge behavior for a given key in a module and let users of that module specify overrides in the environment layer.

As an example, the following configuration defines lookup_options for several keys in a module. One of the keys is overridden at the environment level – the others retain their configuration:

```
# <MYMODULE>/data/common.yaml
```

```
lookup_options:
  mymodule::key1:
    merge:
      strategy: deep
      merge_hash_arrays: true
  mymodule::key2:
    merge: deep
  mymodule::key3:
    merge: deep

# <ENVIRONMENT>/data/common.yaml
lookup_options:
  mymodule::key1:
    merge: deep  # this overrides the merge_hash_arrays true
```

#### Overriding merge behavior in a call to `lookup()`

When you specify a merge behavior as an argument to the lookup function, it overrides the configured merge behavior. For example, with the configuration above:

```
lookup('mymodule::key1', 'strategy' => 'first')
```

The lookup of `'mymodule::key1'` uses strategy `'first'` instead of strategy `'deep'` in the `lookup_options` configuration.

#### Make Hiera return data by casting to a specific data type

To convert values from Hiera backends to rich data values, not representable in YAML or JSON, use the `lookup_options` key `convert_to`, which accepts either a type name or an array of type name and arguments.

When you use `convert_to`, you get automatic type checking. For example, if you specify a `convert_to` using type `"Enum['red', 'blue', 'green']"` and the looked-up value is not one of those strings, it raises an error. You can use this to assert the type when there is not enough type checking in the Puppet code that is doing the lookup.

For types that have a single-value constructor, such as Integer, String, Sensitive, or Timestamp, specify the data type in string form.

For example, to turn a String value into an Integer:

```
mymodule::mykey: "42"
lookup_options:
  mymodule::mykey:
    convert_to: "Integer"
```

To make a value Sensitive:

```
mymodule::mykey: 42
lookup_options:
 mymodule::mykey:
   convert_to: "Sensitive"
```

If the constructor requires arguments, specify type and the arguments in an array. You can also specify it this way when a data type constructor takes optional arguments.

For example, to convert a string ("042") to an Integer with explicit decimal (base 10) interpretation of the string:

```
mymodule::mykey: "042"
lookup_options:
  mymodule::mykey:
    convert_to:
```

```
        - "Integer"
        - 10
```

The default would interpret the leading 0 to mean an octal value (octal 042 is decimal 34):

To turn a non-Array value into an Array:

```
mymodule::mykey: 42
lookup_options:
 mymodule::mykey:
   convert_to:
     - "Array"
     - true
```

**Related information**

Automatic lookup of class parameters on page 419
Puppet looks up the values for class parameters in Hiera, using the fully qualified name of the parameter (`myclass::parameter_one`) as a lookup key.

## Use a regular expression in `lookup_options`

You can use regular expressions in `lookup_options` to configure merge behavior for many lookup keys at the same time.

A regular expression key such as `'^profile::(.*)::users$'` sets the merge behavior for `profile::server::users, profile::postgresql::users`, `profile::jenkins::master::users`. Regular expression lookup options use Puppet's regular expression support, which is based on Ruby's regular expressions.

To use a regular expression in `lookup_options`:

1. Write the pattern as a quoted string. Do not use the Puppet language's forward-slash `(/.../)` regular expression delimiters.
2. Begin the pattern with the start-of-line metacharacter (`^`, also called a carat). If ^ isn't the first character, Hiera treats it as a literal key name instead of a regular expression.
3. If this data source is in a module, follow `^` with the module's namespace: its full name, plus the `::` namespace separator. For example, all regular expression lookup options in the `ntp` module must start with `^ntp::`. Starting with anything else results in an error.

The merge behavior you set for that pattern applies to all lookup keys that match it. In cases where multiple lookup options could apply to the same key, Hiera resolves the conflict. For example, if there's a literal (not regular expression) option available, Hiera uses it. Otherwise, Hiera uses the first regular expression that matches the lookup key, using the order in which they appear in the module code.

**Note:** `lookup_options` are assembled with a hash merge, which puts keys from lower priority data sources before those from higher priority sources. To override a module's regular expression configured merge behavior, use the exact same regular expression string in your environment data, so that it replaces the module's value. A slightly different regular expression won't work because the lower-priority regular expression goes first.

## Interpolation

In Hiera you can insert, or interpolate, the value of a variable into a string, using the syntax `%{variable}`.

Hiera uses interpolation in two places:

- Hierarchies: you can interpolate variables into the `path`, `paths`, `glob`, `globs`, `uri`, `uris`, `datadir`, `mapped_paths`, and `options` of a hierarchy level. This lets each node get a customized version of the hierarchy.

- Data: you can use interpolation to avoid repetition. This takes one of two forms:
  - If some value always involves the value of a fact (for example, if you need to specify a mail server and you have one predictably-named mail server per domain), reference the fact directly instead of manually transcribing it.
  - If multiple keys need to share the same value, write it out for one of them and reuse it for the rest with the `lookup` or `alias` interpolation functions. This makes it easier to keep data up to date, as you only need to change a given value in one place.

### Interpolation token syntax

Interpolation tokens consist of the following:

- A percent sign (%)
- An opening curly brace ({)
- One of:
  - A variable name, optionally using key.subkey notation to access a specific member of a hash or array.
  - An interpolation function and its argument.
- A closing curly brace (}).

For example, `%{trusted.certname}` or `%{alias("users")}`.

Hiera interpolates values of Puppet data types and converts them to strings. Note that the exception to this is when using an alias. If the alias is the only thing present, then its value is not converted.

In YAML files, any string containing an interpolation token must be enclosed in quotation marks.

**Note:** Unlike the Puppet interpolation tokens, you can't interpolate an arbitrary expression.

Related topics: Puppet's data types, Puppet's rules for interpolating non-string values.

## Interpolate a Puppet variable

The most common thing to interpolate is the value of a Puppet top scope variable.

The `facts` hash, `trusted` hash, and `server_facts` hash are the most useful variables to Hiera and behave predictably.

**Note:** If you have a hierarchy level that needs to reference the name of a node, get the node's name by using `trusted.certname`. To reference a node's environment, use `server_facts.environment`.

Avoid using local variables, namespaced variables from classes (unless the class has already been evaluated), and Hiera-specific pseudo-variables (pseudo-variables are not supported in Hiera 5).

If you are using Hiera 3 pseudo-variables, see Puppet variables passed to Hiera.

Puppet makes facts available in two ways: grouped together in the `facts` hash ( `$facts['networking']`), and individually as top-scope variables ( `$networking`).

When you use individual fact variables, specify the (empty) top-scope namespace for them, like this:

- `%{::networking}`

Not like this:

- `%{networking}`

**Note:** The individual fact names aren't protected the way `$facts` is, and local scopes can set unrelated variables with the same names. In most of Puppet, you don't have to worry about unknown scopes overriding your variables, but in Hiera you do.

To interpolate a Puppet variable:

Use the name of the variable, omitting the leading dollar sign (`$`). Use the Hiera key.subkey notation to access a member of a data structure. For example, to interpolate the value of `$facts['networking']['domain']` write: `smtpserver: "mail.%{facts.networking.domain}"`

For more information, see facts, environments.

#### Related information

Access hash and array elements using a key.subkey notation on page 422
Access hash and array members in Hiera using a `key.subkey` notation.

## Interpolation functions

In Hiera data sources, you can use interpolation functions to insert non-variable values. These aren't the same as Puppet functions; they're only available in Hiera interpolation tokens.

**Note:** You cannot use interpolation functions in `hiera.yaml.` They're only for use in data sources.

There are five interpolation functions:

- `lookup` - looks up a key using Hiera, and interpolates the value into a string
- `hiera` - a synonym for `lookup`
- `alias` - looks up a key using Hiera, and uses the value as a replacement for the enclosing string. The result has the same data type as what the aliased key has - no conversion to string takes place if the value is exactly one alias
- `literal` - a way to write a literal percent sign (`%`) without accidentally interpolating something
- `scope` - an alternate way to interpolate a variable. Not generally useful

### The `lookup` and `hiera` function

The `lookup` and `hiera` interpolation functions look up a key and return the resulting value. The result of the lookup must be a string; any other result causes an error. The `hiera` interpolation functions look up a key and return the resulting value. The result of the lookup must be a string; any other result causes an error.

This is useful in the Hiera data sources. If you need to use the same value for multiple keys, you can assign the literal value to one key, then call lookup to reuse the value elsewhere. You can edit the value in one place to change it everywhere it's used.

For example, suppose your WordPress profile needs a database server, and you're already configuring that hostname in data because the MySQL profile needs it. You could write:

```
# in location/pdx.yaml:
profile::mysql::public_hostname: db-server-01.pdx.example.com

# in location/bfs.yaml:
profile::mysql::public_hostname: db-server-06.belfast.example.com

# in common.yaml:
profile::wordpress::database_server:
 "%{lookup('profile::mysql::public_hostname')}"
```

The value of `profile::wordpress::database_server` is always the same as `profile::mysql::public_hostname`. Even though you wrote the WordPress parameter in the `common.yaml` data, it's location-specific, as the value it references was set in your per-location data files.

The value referenced by the lookup function can contain another call to lookup; if you accidentally make an infinite loop, Hiera detects it and fails.

**Note:** The `lookup` and `hiera` interpolation functions aren't the same as the Puppet functions of the same names. They only take a single argument.

### The `alias` function

The `alias` function lets you use reuse Hash, Array, or Boolean values.

When you interpolate `alias` in a string, Hiera replaces that entire string with the aliased value, using its original data type. For example:

```
original:
  - 'one'
  - 'two'
aliased: "%{alias('original')}"
```

A lookup of original and a lookup of aliased would both return the value `['one', 'two']`.

When you use the `alias` function, its interpolation token must be the only text in that string. For example, the following would be an error:

```
aliased: "%{alias('original')} - 'three'"
```

**Note:** A lookup resulting in an interpolation of `alias` referencing a non-existant key returns an empty string, not a Hiera "not found" condition.

### The `literal` function

The `literal` interpolation function lets you escape a literal percent sign (%) in Hiera data, to avoid triggering interpolation where it isn't wanted.

This is useful when dealing with Apache config files, for example, which might include text such as `%{SERVER_NAME}`. For example:

```
server_name_string: "%{literal('%')}{SERVER_NAME}"
```

The value of `server_name_string` would be `%{SERVER_NAME}`, and Hiera would not attempt to interpolate a variable named `SERVER_NAME`.

The only legal argument for `literal` is a single `%` sign.

### The scope function

The `scope` interpolation function interpolates variables.

It works identically to variable interpolation. The functions argument is the name of a variable.

The following two values would be identical:

```
smtpserver: "mail.%{facts.domain}"
smtpserver: "mail.%{scope('facts.domain')}"
```

### Using interpolation functions

To use an interpolation function to insert non-variable values, write:

1. The name of the function.
2. An opening parenthesis.
3. One argument to the function, enclosed in single or double quotation marks.
4. Use the opposite of what the enclosing string uses: if it uses single quotation marks, use double quotation marks.
5. A closing parenthesis.

For example:

```
wordpress::database_server: "%{lookup('instances::mysql::public_hostname')}"
```

**Note:** There must be no spaces between these elements.

# Looking up data with Hiera

## Automatic lookup of class parameters

Puppet looks up the values for class parameters in Hiera, using the fully qualified name of the parameter (`myclass::parameter_one`) as a lookup key.

Most classes need configuration, and you can specify them as parameters to a class as this looks up the needed data if not directly given when the class is included in a catalog. There are several ways Puppet sets values for class parameters, in this order:

1. If you're doing a resource-like declaration, Puppet uses parameters that are explicitly set (if explicitly setting undef, a looked up value or default is used).
2. Puppet uses Hiera, using `<CLASS NAME>::<PARAMETER NAME>` as the lookup key. For example, it looks up `ntp::servers` for the `ntp` class's `$servers parameter`.
3. If a parameter still has no value, Puppet uses the default value from the parameter's default value expression in the class's definition.
4. If any parameters have no value and no default, Puppet fails compilation with an error.

For example, you can set servers for the `NTP` class like this:

```
# /etc/puppetlabs/code/production/data/nodes/web01.example.com.yaml
---
ntp::servers:
  - time.example.com
  - 0.pool.ntp.org
```

The best way to manage this is to use the roles and profiles method, which allows you to store a smaller amount of meaningful data in Hiera.

**Note:** Automatic lookup of class parameters uses the "first" merge method by default. You cannot change the default. If you want to get deep merges on keys, use the `lookup_options` feature.

This feature is often referred to as Automatic Parameter Lookup (APL).

## The Puppet `lookup` function

The `lookup` function uses Hiera to retrieve a value for a given key.

By default, the `lookup` function returns the first value found and fails compilation if no values are available. You can also configure the lookup function to merge multiple values into one.

When looking up a key, Hiera searches up to four hierarchy layers of data, in the following order:

1. Global hierarchy.
2. The current environment's hierarchy.
3. The indicated module's hierarchy, if the key is of the form `<MODULE NAME>::<SOMETHING>`.
4. If not found and the module's hierarchy has a `default_hierarchy` entry in its `hiera.yaml` — the lookup is repeated if steps 1-3 did not produce a value.

**Note:** Hiera checks the global layer before the environment layer. If no global `hiera.yaml` file has been configured, Hiera defaults are used. If you do not want it to use the defaults, you can create an empty `hiera.yaml` file in `/etc/puppetlabs/puppet/hiera.yaml.`

Default global hiera.yaml is installed at `/etc/puppetlabs/puppet/hiera.yaml`.

### Arguments

You must provide the key's name. The other arguments are optional.

You can combine these arguments in the following ways:

- `lookup( <NAME>, [<VALUE TYPE>], [<MERGE BEHAVIOR>], [<DEFAULT VALUE>] )`
- `lookup( [<NAME>], <OPTIONS HASH> )`
- `lookup( as above ) |$key| { <VALUE> } # lambda returns a default value`

Arguments in `[square brackets]` are optional.

**Note:** Giving a hash of options containing `default_value` at the same time as giving a lambda means that the lambda wins. The rationale for allowing this is that you might be using the same hash of options multiple times, and you might want to override the production of the default value. A `default_values_hash` wins over the lambda if it has a value for the looked up key.

Arguments accepted by lookup:

- `<NAME>` (String or Array) - The name of the key to look up. This can also be an array of keys. If Hiera doesn't find anything for the first key, it tries with the subsequent ones, only resorting to a default value if none of them succeed.
- `<VALUE TYPE>` (data Type) - A data type that must match the retrieved value; if not, the lookup (and catalog compilation) fails. Defaults to `Data` which accepts any normal value.
- `<MERGE BEHAVIOR>` (String or Hash; see Merge behaviors) - Whether and how to combine multiple values. If present, this overrides any merge behavior specified in the data sources. Defaults to no value; Hiera uses merge behavior from the data sources if present, otherwise it does a first-found lookup.
- `<DEFAULT VALUE>` (any normal value) - If present, lookup returns this when it can't find a normal value. Default values are never merged with found values. Like a normal value, the default must match the value type. Defaults to no value; if Hiera can't find a normal value, the lookup (and compilation) fails.
- `<OPTIONS HASH>` (Hash) - Alternate way to set the arguments above, plus some less common additional options. If you pass an options hash, you can't combine it with any regular arguments (except `<NAME>`). An options hash can have the following keys:
  - `'name'` - Same as `<NAME>` (argument 1). You can pass this as an argument or in the hash, but not both.
  - `'value_type'` - Same as `<VALUE TYPE>`.
  - `'merge'` - Same as `<MERGE BEHAVIOR>`.
  - `'default_value'` - Same as `<DEFAULT VALUE>` .
  - `'default_values_hash'` (Hash) - A hash of lookup keys and default values. If Hiera can't find a normal value, it checks this hash for the requested key before giving up. You can combine this with `default_value` or a lambda, which is used if the key isn't present in this hash. Defaults to an empty hash.
  - `'override'` (Hash) - A hash of lookup keys and override values. Puppet checks for the requested key in the overrides hash first. If found, it returns that value as the final value, ignoring merge behavior. Defaults to an empty hash.
  - `lookup` - can take a lambda, which must accept a single parameter. This is yet another way to set a default value for the lookup; if no results are found, Puppet passes the requested key to the lambda and use its result as the default value.

### Merge behaviors

Hiera uses a hierarchy of data sources, and a given key can have values in multiple sources. Hiera can either return the first value it finds, or continue to search and merge all the values together. When Hiera searches, it first searches the global layer, then the environment layer, and finally the module layer — where it only searches in modules that have a matching namespace. By default (unless you use one of the merge strategies) it is priority/"first found wins", in which case the search ends as soon as a value is found.

**Note:** Data sources can use the `lookup_options` metadata key to request a specific merge behavior for a key. The lookup function uses that requested behavior unless you specify one.

Examples:

Default values for a lookup:

(Still works, but deprecated)

```
hiera('some::key', 'the default value')
```

(Recommended)

```
lookup('some::key', undef, undef, 'the default value')
```

Look up a key and returning the first value found:

```
lookup('ntp::service_name')
```

A unique merge lookup of class names, then adding all of those classes to the catalog:

```
lookup('classes', Array[String], 'unique').include
```

A deep hash merge lookup of user data, but letting higher priority sources remove values by prefixing them with:

```
lookup( { 'name'  => 'users',
          'merge' => {
            'strategy'        => 'deep',
            'knockout_prefix' => '--',
          },
})
```

## The `puppet lookup` command

The `puppet lookup` command is the command line interface (CLI) for Puppet's lookup function.

The `puppet lookup` command lets you do Hiera lookups from the command line. You must run it on a node that has a copy of your Hiera data. You can log into a Puppet Server node and run `puppet lookup` with `sudo`.

The most common version of this command is:

```
puppet lookup <KEY> --node <NAME> --environment <ENV> --explain
```

The `puppet lookup` command searches your Hiera data and returns a value for the requested lookup key, so you can test and explore your data. It replaces the `hiera` command. Hiera relies on a node's facts to locate the relevant data sources. By default, `puppet lookup` uses facts from the node you run the command on, but you can get data for any other node with the `--node NAME` option. If possible, the lookup command uses the requested node's real stored facts from PuppetDB. If PuppetDB is not configured or you want to provide other fact values, pass facts from a JSON or YAML file with the `--facts FILE` option.

**Note:** The `puppet lookup` command replaces the `hiera` command.

### Examples

To look up `key_name` using the Puppet Server node’s facts:

```
$ puppet lookup key_name
```

To look up `key_name` with `agent.local`'s facts:

```
$ puppet lookup --node agent.local key_name
```

To get the first value found for `key_name_one` and `key_name_two` with `agent.local`'s facts while merging values and knocking out the prefix 'example' while merging:

```
puppet lookup --node agent.local --merge deep --knock-out-prefix example
 key_name_one key_name_two
```

To lookup `key_name` with `agent.local`'s facts, and return a default value of `0` if nothing is found:

```
puppet lookup --node agent.local --default 0 key_name
```

To see an explanation of how the value for `key_name` is found, using `agent.local` facts:

```
puppet lookup --node agent.local --explain key_name
```

#### Options

The `puppet lookup` command has the following command options:

- `--help`: Print a usage message.
- `--explain`: Explain the details of how the lookup was performed and where the final value came from, or the reason no value was found. Useful when debugging Hiera data. If `--explain` isn't specified, lookup exits with 0 if a value was found and 1 if not. With `--explain`, lookup always exits with 0 unless there is a major error. You can provide multiple lookup keys to this command, but it only returns a value for the first found key, omitting the rest.
- `--node <NODE-NAME>`: Specify which node to look up data for; defaults to the node where the command is run. The purpose of Hiera is to provide different values for different nodes; use specific node facts to explore your data. If the node where you're running this command is configured to talk to PuppetDB, the command uses the requested node's most recent facts. Otherwise, override facts with the '--facts' option.
- `--facts <FILE>`: Specify a JSON or YAML file that contains key-value mappings to override the facts for this lookup. Any facts not specified in this file maintain their original value.
- `--environment <ENV>`: Specify an environment. Different environments can have different Hiera data.
- `--merge first/unique/hash/deep`: Specify the merge behavior, overriding any merge behavior from the data's `lookup_options`.
- `--knock-out-prefix <PREFIX-STRING>`: Used with 'deep' merge. Specifies a prefix to indicate a value should be removed from the final result.
- `--sort-merged-arrays`: Used with 'deep' merge. When this flag is used, all merged arrays are sorted.
- `--merge-hash-arrays`: Used with the 'deep' merge strategy. When this flag is used, hashes within arrays are deep-merged with their counterparts by position.
- `--explain-options`: Explain whether a `lookup_options` hash affects this lookup, and how that hash was assembled. (`lookup_options` is how Hiera configures merge behavior in data.)
- `--default <VALUE>`: A value to return if Hiera can't find a value in data. Useful for emulating a call to the `lookup function that includes a default.
- `--type <TYPESTRING>`: Assert that the value has the specified type. Useful for emulating a call to the `lookup` function that includes a data type.
- `--compile`: Perform a full catalog compilation prior to the lookup. If your hierarchy and data only use the `$facts`, `$trusted`, and `$server_facts` variables, you don't need this option. If your Hiera configuration uses arbitrary variables set by a Puppet manifest, you need this to get accurate data. The `lookup` command doesn't cause catalog compilation unless this flag is given.
- `--render-as s/json/yaml/binary/msgpack`: Specify the output format of the results; `s` means plain text. The default when producing a value is `yaml` and the default when producing an explanation is `s`.

## Access hash and array elements using a `key.subkey` notation

Access hash and array members in Hiera using a `key.subkey` notation.

You can access hash and array elements when doing the following things:

- Interpolating variables into `hiera.yaml` or a data file. Many of the most commonly used variables, for example `facts` and `trusted`, are deeply nested data structures.
- Using the `lookup` function or the `puppet lookup` command. If the value of `lookup('some_key')` is a hash or array, look up a single member of it by using `lookup('some_key.subkey')`.
- Using interpolation functions that do Hiera lookups, for example `lookup` and `alias`.

To access a single member of an array or hash:

Use the name of the value followed by a period (`.`) and a subkey.

- If the value is an array, the subkey must be an integer, for example: `users.0` returns the first entry in the `users` array.
- If the value is a hash, the subkey must be the name of a key in that hash, for example, `facts.os`.
- To access values in nested data structures, you can chain subkeys together. For example, because the value of `facts.system_uptime` is a hash, you can access its hours key with `facts.system_uptime.hours`.

### Hiera dotted notation

The Hiera dotted notation does not support arbitrary expressions for subkeys; only literal keys are valid.

A hash can include literal dots in the text of a key. For example, the value of `$trusted['extensions']` is a hash containing any certificate extensions for a node, but some of its keys can be raw OID strings like `'1.3.6.1.4.1.34380.1.2.1'`. You can access those values in Hiera with the `key.subkey` notation, but you must put quotation marks — single or double — around the affected subkey. If the entire compound key is quoted (for example, as required by the lookup interpolation function), use the other kind of quote for the subkey, and escape quotes (as needed by your data file format) to ensure that you don't prematurely terminate the whole string.

For example:

```
aliased_key: "%{lookup('other_key.\"dotted.subkey\"')}"
# Or:
aliased_key: "%{lookup(\"other_key.'dotted.subkey'\")}"
```

**Note:** Using extra quotes prevents digging into dotted keys. For example, if the lookup key contains a dot (`.`) then the entire key must be enclosed within single quotes within double quotes, for example, `lookup("'has.dot'")`.

## Writing new data backends

You can extend Hiera to look up values in data sources, for example, a PostgreSQL database table, a custom web app, or a new kind of structured data file.

To teach Hiera how to talk to other data sources, write a custom backend.

**Important:** Writing a custom backend is an advanced topic. Before proceeding, make sure you really need it. It is also worth asking the puppet-dev mailing list or Slack channel to see whether there is one you can re-use, rather than starting from scratch.

### Custom backends overview

A backend is a custom Puppet function that accepts a particular set of arguments and whose return value obeys a particular format. The function can do whatever is necessary to locate its data.

A backend function uses the modern Ruby functions API or the Puppet language. This means you can use different versions of a Hiera backend in different environments, and you can distribute Hiera backends in Puppet modules.

Different types of data have different performance characteristics. To make sure Hiera performs well with every type of data source, it supports three types of backends: `data_hash`, `lookup_key`, and `data_dig`.

#### data_hash

For data sources where it's inexpensive, performance-wise, to read the entire contents at one time, like simple files on disk. We suggest using the `data_hash` backend type if:

- The cache is alive for the duration of one compilation
- The data is small
- The data can be retrieved all at one time
- Most of the data gets used
- The data is static

For more information, see data_hash backends.

#### lookup_key

For data sources where looking up a key is relatively expensive, performance-wise, like an HTTPS API. We suggest using the `lookup_key` backend type if:

- The data set is big, but only a small portion is used
- The result can vary during the compilation

The `hiera-eyaml` backend is a `lookup_key` function, because decryption tends to affect performance; as a given node uses only a subset of the available secrets, it makes sense to decrypt only on-demand.

For more information, see lookup_key backends.

#### data_dig

For data sources that can access arbitrary elements of hash or array values before passing anything back to Hiera, like a database.

For more information, see data_dig backends.

### The `RichDataKey` and `RichData` types

To simplify backend function signatures, you can use two extra data type aliases: `RichDataKey` and `RichData`. These are only available to backend functions called by Hiera; normal functions and Puppet code can not use them.

For more information, see custom Puppet functions, the modern Ruby functions API.

## data_hash backends

A `data_hash` backend function reads an entire data source at one time, and returns its contents as a hash.

The built-in YAML, JSON, and HOCON backends are all `data_hash` functions. You can view their source on GitHub:

- yaml_data.rb
- json_data.rb
- hocon_data.rb

### Arguments

Hiera calls a `data_hash` function with two arguments:

- A hash of options
  - The options hash contains a `path` when the entry in hiera.yaml is using `path`, `paths`, `glob`, `globs`, or `mapped_paths`, and the backend receives one call per path to an existing file. When the entry in `hiera.yaml` is using `uri` or `uris`, the options hash has a `uri` key, and the backend function is called one time per given uri. When `uri` or `uris` are used, Hiera does not perform an existence check. It is up to the function to type the options parameter as wanted.
- A `Puppet::LookupContext` object

### Return type

The function must either call the context object's `not_found` method, or return a hash of lookup keys and their associated values. The hash can be empty.

Puppet language example signature:

```
function mymodule::hiera_backend(
  Hash                  $options,
  Puppet::LookupContext $context,
)
```

Ruby example signature:

```
dispatch :hiera_backend do
  param 'Hash', :options
  param 'Puppet::LookupContext', :context
end
```

The returned hash can include the `lookup_options` key to configure merge behavior for other keys. See Configuring merge behavior in Hiera data for more information. Values in the returned hash can include Hiera interpolation tokens like `%{variable}` or `%{lookup('key')}`; Hiera interpolates values as needed. This is a significant difference between `data_hash` and the other two backend types; `lookup_key` and `data_dig` functions have to explicitly handle interpolation.

### Related information

Configure merge behavior in Hiera data

## `lookup_key` backends

A `lookup_key` backend function looks up a single key and returns its value. For example, the built-in `hiera_eyaml` backend is a `lookup_key` function.

You can view its source on Git at eyaml_lookup_key.rb.

### Arguments

Hiera calls a `lookup_key` function with three arguments:

- A key to look up.
- A hash of options.
- A `Puppet::LookupContext` object.

### Return type

The function must either call the context object's `not_found` method, or return a value for the requested key. It can return undef as a value.

Puppet language example signature:

```
function mymodule::hiera_backend(
  Variant[String, Numeric] $key,
  Hash                     $options,
  Puppet::LookupContext    $context,
)
```

Ruby example signature:

```
dispatch :hiera_backend do
  param 'Variant[String, Numeric]', :key
  param 'Hash', :options
  param 'Puppet::LookupContext', :context
```

```
end
```

A `lookup_key` function can return a hash for the the `lookup_options` key to configure merge behavior for other keys. See Configuring merge behavior in Hiera data for more information. To support Hiera interpolation tokens, for example, `%{variable}` or `%{lookup('key')}` in your data, call `context.interpolate` on your values before returning them.

#### Related information

Interpolation on page 415
In Hiera you can insert, or interpolate, the value of a variable into a string, using the syntax `%{variable}`.

Hiera calling conventions for backend functions on page 427
Hiera uses the following conventions when calling backend functions.

## `data_dig` backends

A `data_dig` backend function is similar to a `lookup_key` function, but instead of looking up a single key, it looks up a single sequence of keys and subkeys.

Hiera lets you look up individual members of hash and array values using `key.subkey` notation. Use `data_dig` types in cases where:

- Lookups are relatively expensive.
- The data source knows how to extract elements from hash and array values.
- Users are likely to pass `key.subkey` requests to the `lookup` function to access subsets of large data structures.

#### Arguments

Hiera calls a `data_dig` function with three arguments:

- An array of lookup key segments, made by splitting the requested lookup key on the dot (`.`) subkey separator. For example, a lookup for `users.dbadmin.uid` results in `['users', 'dbadmin', 'uid']`. Positive base-10 integer subkeys (for accessing array members) are converted to Integer objects, but other number subkeys remain as strings.
- A hash of options.
- A `Puppet::LookupContext` object.

#### Return type

The function must either call the context object's `not_found` method, or return a value for the requested sequence of key segments. Note that returning undef (nil in Ruby) means that the key was found but that the value for that key was specified to be undef. Puppet language example signature:

```
function mymodule::hiera_backend(
  Array[Variant[String, Numeric]] $segments,
  Hash                            $options,
  Puppet::LookupContext           $context,
)
```

Ruby example signature:

```
dispatch :hiera_backend do
  param 'Array[Variant[String, Numeric]]', :segments
  param 'Hash', :options
  param 'Puppet::LookupContext', :context
end
```

A `data_dig` function can return a hash for the the `lookup_options` key to configure merge behavior for other keys. See Configuring merge behavior in Hiera data for more info.

To support Hiera interpolation tokens like `%{variable}` or `%{lookup('key')}` in your data, call `context.interpolate` on your values before returning them.

**Related information**

Configure merge behavior in Hiera data

Access hash and array elements using a key.subkey notation on page 422
Access hash and array members in Hiera using a `key.subkey` notation.

## Hiera calling conventions for backend functions

Hiera uses the following conventions when calling backend functions.

Hiera calls `data_hash` one time per data source, calls `lookup_key` functions one time per data source for every unique key lookup, and calls `data_dig` functions one time per data source for every unique sequence of key segments.

However, a given hierarchy level can refer to multiple data sources with the `path`, `paths`, `uri`, `uris`, `glob`, and `globs` settings. Hiera handles each hierarchy level as follows:

- If the `path`, `paths`, `glob`, or `globs` settings are used, Hiera determines which files exist and calls the function one time for each. If no files were found, the function is not be called.
- If the `uri` or `uris` settings are used, Hiera calls the function one time per URI.
- If none of those settings are used, Hiera calls the function one time.

Hiera can call a function again for a given data source, if the inputs change. For example, if `hiera.yaml` interpolates a local variable in a file path, Hiera calls the function again for scopes where that variable has a different value. This has a significant performance impact, so you must interpolate only facts, trusted facts, and server facts in the hierarchy.

## The options hash

Hierarchy levels are configured in the `hiera.yaml` file. When calling a backend function, Hiera passes a modified version of that configuration as a hash.

The options hash can contain (depending on whether `path`, `glob`, `uri`, or`mapped_paths` have been set) the following keys:

- `path` - The absolute path to a file on disk. It is present only if `path`, `paths`, `glob`, `globs`, or `mapped_paths` is present in the hierarchy. Hiera never calls the function unless the file is present.
- `uri` - A uri that your function can use to locate a data source. It is present only if `uri` or `uris` is present in the hierarchy. Hiera does not verify the URI before passing it to the function.
- Every key from the hierarchy level's `options` setting. List any options your backend requires or accepts. The `path` and `uri` keys are reserved.

**Note:** If your backend uses data files, use the context object's `cached_file_data` method to read them.

For example, the following hierarchy level in `hiera.yaml` results in several different options hashes, depending on such things as the current node's facts and whether the files exist:

```
- name: "Secret data: per-node, per-datacenter, common"
    lookup_key: eyaml_lookup_key # eyaml backend
    datadir: data
    paths:
      - "secrets/nodes/%{trusted.certname}.eyaml"
      - "secrets/location/%{facts.whereami}.eyaml"
      - "common.eyaml"
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
      pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem
```

The various hashes would all be similar to this:

```
{
  'path' => '/etc/puppetlabs/code/environments/production/data/secrets/
nodes/web01.example.com.eyaml',
  'pkcs7_private_key' => '/etc/puppetlabs/puppet/eyaml/
private_key.pkcs7.pem',
  'pkcs7_public_key' => '/etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem'
}
```

In your function's signature, you can validate the options hash by using the Struct data type to restrict its contents. In particular, note you can disable all of the `path`, `paths`, `glob`, and `globs` settings for your backend by disallowing the `path` key in the options hash.

For more information, see the Struct data type.

**Related information**

Configure merge behavior in Hiera data

Configuring a hierarchy level: hiera-eyaml on page 407
Hiera 5 ( Puppet 4.9.3 and later) includes a native interface for the Hiera eyaml extension, which keeps data encrypted on disk but lets Puppet read it during catalog compilation.

Interpolation on page 415
In Hiera you can insert, or interpolate, the value of a variable into a string, using the syntax `%{variable}`.

## The `Puppet::LookupContext` object and methods

To support caching and other backends needs, Hiera provides a `Puppet::LookupContext` object.

In Ruby functions, the context object is a normal Ruby object of class `Puppet::LookupContext`, and you can call methods with standard Ruby syntax, for example `context.not_found`.

In Puppet language functions, the context object appears as the special data type `Puppet::LookupContext`, that has methods attached.You can call the context's methods using Puppet's chained function call syntax with the method name instead of a normal function call syntax, for example, `$context.not_found`. For methods that take a block, use Puppet's lambda syntax (parameters outside block) instead of Ruby's block syntax (parameters inside block).

### `not_found()`

Tells Hiera to halt this lookup and move on to the next data source. Call this method when your function cannot find a matching key or a given lookup. This method returns no value.

For `data_hash` backends, return an empty hash. The empty hash results in `not_found`, and prevents further calls to the provider. Missing data sources are not an issue when using `path`, `paths`, `glob`, or `globs`, but are important for backends that locate their own data sources.

For `lookup_key` and `data_dig` backends, use `not_found` when a requested key is not present in the data source or the data source does not exist. Do not return `undef` or `nil` for missing keys, as these are legal values that can be set in data.

### `interpolate(value)`

Returns the provided value, but with any Hiera interpolation tokens (`%{variable}` or `%{lookup('key')}`) replaced by their value. This lets you opt-in to allowing Hiera-style interpolation in your backend's data sources. It works recursively on arrays and hashes. Hashes can interpolate into both keys and values.

In `data_hash` backends, support for interpolation is built in, and you do not need to call this method.

In `lookup_key` and `data_dig` backends, call this method if you want to support interpolation.

### `environment_name()`

Returns the name of the environment, regardless of layer.

### `module_name()`

Returns the name of the module whose `hiera.yaml` called the function. Returns `undef` (in Puppet) or `nil` (in Ruby) if the function was called by the global or environment layer.

### `cache(key, value)`

Caches a value, in a per-data-source private cache. It also returns the cached value.

On future lookups in this data source, you can retrieve values by calling `cached_value(key)`. Cached values are immutable, but you can replace the value for an existing key. Cache keys can be anything valid as a key for a Ruby hash, including `nil`.

For example, on its first invocation for a given YAML file, the built-in `eyaml_lookup_key` backend reads the whole file and caches it, and then decrypts only the specific value that was requested. On subsequent lookups into that file, it gets the encrypted value from the cache instead of reading the file from disk again. It also caches decrypted values so that it won't have to decrypt again if the same key is looked up repeatedly.

The cache is useful for storing session keys or connection objects for backends that access a network service.

Each `Puppet::LookupContext` cache lasts for the duration of the current catalog compilation. A node can't access values cached for a previous node.

Hiera creates a separate cache for each combination of inputs for a function call, including inputs like `name` that are configured in `hiera.yaml` but not passed to the function. Each hierarchy level has its own cache, and hierarchy levels that use multiple paths have a separate cache for each path.

If any inputs to a function change, for example, a path interpolates a local variable whose value changes between lookups, Hiera uses a fresh cache.

### `cache_all(hash)`

Caches all the key-value pairs from a given hash. Returns `undef` (in Puppet) or `nil` (in Ruby).

### `cached_value(key)`

Returns a previously cached value from the per-data-source private cache. Returns `undef` or `nil` if no value with this name has been cached.

### `cache_has_key(key)`

Checks whether the cache has a value for a given key yet. Returns `true` or `false`.

### `cached_entries()`

Returns everything in the `per-data-source` cache as an iterable object. The returned object is not a hash. If you want a hash, use `Hash($context.all_cached())` in the Puppet language or `Hash[context.all_cached()]` in Ruby.

### `cached_file_data(path)`

Puppet syntax:

```
cached_file_data(path) |content| { ... }
```

Ruby syntax:

```
cached_file_data(path) {|content| ...}
```

For best performance, use this method to read files in Hiera backends.

`cached_file_data(path) {|content| ...}` returns the content of the specified file as a string. If an optional block is provided, it passes the content to the block and returns the block's return value. For example, the built-in JSON backend uses a block to parse JSON and return a hash:

```
context.cached_file_data(path) do |content|
      begin
        JSON.parse(content)
      rescue JSON::ParserError => ex
        # Filename not included in message, so we add it here.
        raise Puppet::DataBinding::LookupError, "Unable to parse (#{path}):
 #{ex.message}"
      end
    end
```

On repeated access to a given file, Hiera checks whether the file has changed on disk. If it hasn't, Hiera uses cached data instead of reading and parsing the file again.

This method does not use the same `per-data-source` caches as `cache(key, value)` and similar methods. It uses a separate cache that lasts across multiple catalog compilations, and is tied to Puppet Server's environment cache.

Because the cache can outlive a given node's catalog compilation, do not do any node-specific pre-processing (like calling `context.interpolate`) in this method's block.

### explain() { 'message' }

Puppet syntax:

`explain() || { 'message' }`

Ruby syntax:

`explain() { 'message' }`

In both Puppet and Ruby, the provided block must take zero arguments.

`explain() { 'message' }` adds a message, which appears in debug messages or when using puppet lookup --explain. The block provided to this function must return a string.

The explain method is useful for complex lookups where a function tries several different things before arriving at the value. The built-in backends do not use the explain method, and they still have relatively verbose explanations. This method is for when you need to provide even more detail.

Hiera never executes the explain block unless explain is enabled.

## Upgrading to Hiera 5

Upgrading to Hiera 5 offers some major advantages. A real environment data layer means changes to your hierarchy are now routine and testable, using multiple backends in your hierarchy is easier and you can make a custom backend.

**Note:** If you're already a Hiera user, you can use your current code with Hiera 5 without any changes to it. Hiera 5 is fully backward-compatible with Hiera 3. You can even start using some Hiera 5 features—like module data—without upgrading anything.

Hiera 5 uses the same built-in data formats as Hiera 3. You don't need to do mass edits of any data files.

Updating your code to take advantage of Hiera 5 features involves the following tasks:

| Task | Benefit |
| --- | --- |
| Enable the environment layer, by giving each environment its own `hiera.yaml` file.<br><br>**Note:** Enabling the environment layer takes the most work, but yields the biggest benefits. Focus on that first, then do the rest at your own pace. | Future hierarchy changes are cheap and testable. The legacy Hiera functions (`hiera`, `hiera_array`, `hiera_hash`, and `hiera_include`) gain full Hiera 5 powers in any migrated environment, only if there is a `hiera.yaml` in the environment root. |
| Convert your global `hiera.yaml` file to the version 5 format. | You can use new Hiera 5 backends at the global layer. |
| Convert any experimental (version 4) `hiera.yaml` files to version 5. | Future-proof any environments or modules where you used the experimental version of Puppet lookup. |
| In Puppet code, replace legacy Hiera functions (`hiera`, `hiera_array`, `hiera_hash`, and `hiera_include`) with `lookup()`. | Future-proof your Puppet code. |
| Use Hiera for default data in modules. | Simplify your modules with an elegant alternative to the "`params.pp`" pattern. |

### Considerations for hiera-eyaml users

Upgrade now. In Puppet 4.9.3, we added a built-in hiera-eyaml backend for Hiera 5. (It still requires that the `hiera-eyaml` gem be installed.) See the usage instructions in the `hiera.yaml` (v5) syntax reference. This means you can move your existing encrypted YAML data into the environment layer at the same time you move your other data.

### Considerations for custom backend users

Wait for updated backends. You can keep using custom Hiera 3 backends with Hiera 5, but they'll make upgrading more complex, because you can't move legacy data to the environment layer until there's a Hiera 5 backend for it. If an updated version of the backend is coming out soon, wait.

If you're using an off-the-shelf custom backend, check its website or contact its developer. If you developed your backend in-house, read the documentation about writing Hiera 5 backends.

### Considerations for custom `data_binding_terminus` users

Upgrade now, and replace it with a Hiera 5 backend as soon as possible. There's a deprecated `data_binding_terminus` setting in the `puppet.conf` file, which changes the behavior of automatic class parameter lookup. It can be set to `hiera` (normal), `none` (deprecated; disables auto-lookup), or the name of a custom plug-in.

With a custom `data_binding_terminus`, automatic lookup results are different from function-based lookups for the same keys. If you're one of the few who use this feature, you've already had to design your Puppet code to avoid that problem, so it's safe to upgrade your configuration to Hiera 5. But because we've deprecated that extension point, you have to replace your custom terminus with a Hiera 5 backend.

If you're using an off-the-shelf plug-in, such as Jerakia, check its website or contact its developer. If you developed your plug-in in-house, read the documentation about writing Hiera 5 backends.

After you have a Hiera 5 backend, integrate it into your hierarchies and delete the `data_binding_terminus` setting.

### Related information

The Puppet lookup function on page 419
The `lookup` function uses Hiera to retrieve a value for a given key.

Config file syntax on page 403

The `hiera.yaml` file is a YAML file, containing a hash with up to four top-level keys.

Writing new data backends on page 423
You can extend Hiera to look up values in data sources, for example, a PostgreSQL database table, a custom web app, or a new kind of structured data file.

Hiera configuration layers on page 398
Hiera uses three independent layers of configuration. Each layer has its own hierarchy, and they're linked into one super-hierarchy before doing a lookup.

## Enable the environment layer for existing Hiera data

A key feature in Hiera 5 is per-environment hierarchy configuration. Because you probably store data in each environment, local `hiera.yaml` files are more logical and convenient than a single global hierarchy.

You can enable the environment layer gradually. In migrated environments, the legacy Hiera functions switch to Hiera 5 mode — they can access environment and module data without requiring any code changes.

**Note:** Before migrating environment data to Hiera 5, read the introduction to migrating Hiera configurations. In particular, be aware that if you rely on custom Hiera 3 backends, we recommend you upgrade them for Hiera 5 or prepare for some extra work during migration. If your only custom backend is `hiera-eyaml`, continue upgrading — Puppet 4.9.3 and higher include a Hiera 5 eyaml backend. See the usage instructions in the `hiera.yaml` (v5) syntax reference.

In each environment:

1. Check your code for Hiera function calls with "hierarchy override arguments" (as shown later), which cause errors.
2. Add a local `hiera.yaml` file.
3. Update your custom backends if you have them.
4. Rename the data directory to exclude this environment from the global layer. Unmigrated environments still rely on the global layer, which gets checked before the environment layer. If you want to maintain both migrated and unmigrated environments during the migration process, choose a different data directory name for migrated environments. The new name 'data' is a good choice because it is also the new default (unless you are already using 'data', in which case choose a different name and set `datadir` in `hiera.yaml`). This process has no particular time limit and doesn't involve any downtime. After all of your environments are migrated, you can phase out or greatly reduce the global hierarchy.

   **Important:** The environment layer does not support Hiera 3 backends. If any of your data uses a custom backend that has not been ported to Hiera 5, omit those hierarchy levels from the environment config and continue to use the global layer for that data. Because the global layer is checked before the environment layer, it's possible to run into situations where you cannot migrate data to the environment layer yet. For example, if your old `:backends` setting was `[custom_backend, yaml]`, you can do a partial migration, because the custom data was all going before the YAML data anyway. But if `:backends` was `[yaml, custom_backend]`, and you frequently use YAML data to override the custom data, you can't migrate until you have a Hiera 5 version of that custom backend. If you run into a situation like this, get an upgraded backend before enabling the environment layer.
5. Check your Puppet code for classic Hiera functions (`hiera`, `hiera_array`, `hiera_hash`, and `hiera_include`) that are passing the optional hierarchy override argument, and remove the argument.

   In Hiera 5, the hierarchy override argument is an error.

   A quick way to find instances of using this argument is to search for calls with two or more commas. Search your codebase using the following regular expression:

   ```
   hiera(_array|_hash|_include)?\(([^,\)]*,){2,}[^\)]*\)
   ```

   This results in some false positives, but helps find the errors before you run the code.

   Alternatively, continue to the next step and fix errors as they come up. If you use environments for code testing and promotion, you're probably migrating a temporary branch of your control repo first, then pointing some canary nodes at it to make sure everything works as expected. If you think you've never used hierarchy override arguments, you'll be verifying that assumption when you run your canary nodes. If you do find any errors, you can fix them before merging your branch to production, the same way you would with any work-in-progress code.

**Note:** If your environments are similar to each other, you might only need to check for the hierarchy override argument in function calls in one environment. If you find none, likely the others won't have many either.

6. Choose a new data directory name to use in the next two steps. The default data directory name in Hiera 3 was `<ENVIRONMENT>/hieradata`, and the default in Hiera 5 is `<ENVIRONMENT>/data`. If you used the old default, use the new default. If you were already using data, choose something different.
7. Add a Hiera 5 `hiera.yaml` file to the environment.

   Each environment needs a Hiera config file that works with its existing data. If this is the first environment you're migrating, see converting a version 3 `hiera.yaml` to version 5. Make sure to reference the new `datadir` name. If you've already migrated at least one environment, copy the `hiera.yaml` file from a migrated environment and make changes to it if necessary.

   Save the resulting file as `<ENVIRONMENT>/hiera.yaml`. For example, `/etc/puppetlabs/code/environments/production/hiera.yaml`.
8. If any of your data relies on custom backends that have been ported to Hiera 5, install them in the environment. Hiera 5 backends are distributed as Puppet modules, so each environment can use its own version of them.
9. If you use only file-based Hiera 5 backends, move the environment's data directory by renaming it from its old name (`hieradata`) to its new name (`data`). If you use custom file-based Hiera 3 backends, the global layer still needs access to their data, so you need to sort the files: Hiera 5 data moves to the new data directory, and Hiera 3 data stays in the old data directory. When you have Hiera 5 versions of your custom backends, you can move the remaining files to the new `datadir`. If you use non-file backends that don't have a data directory:
   a) Decide that the global hierarchy is the right place for configuring this data, and leave it there permanently.
   b) Do something equivalent to moving the `datadir`; for example, make a new database table for migrated data and move values into place as you migrate environments.
   c) Allow the global and environment layers to use duplicated configuration for this data until the migration is done.
10. Repeat these steps for each environment. If you manage your code by mapping environments to branches in a control repo, you can migrate most of your environments using your version control system's merging tools.
11. After you have migrated the environments that have active node populations, delete the parts of your global hierarchy that you transferred into environment hierarchies.

For more information on mapping environments to branches, see control repo.

### Related information

Enable the environment layer for existing Hiera data on page 432
A key feature in Hiera 5 is per-environment hierarchy configuration. Because you probably store data in each environment, local `hiera.yaml` files are more logical and convenient than a single global hierarchy.

Configuring a hierarchy level: legacy Hiera 3 backends on page 408
If you rely on custom data backends designed for Hiera 3, you can use them in your global hierarchy. They are not supported at the environment or module layers.

Config file syntax on page 403
The `hiera.yaml` file is a YAML file, containing a hash with up to four top-level keys.

Custom backends overview on page 423
A backend is a custom Puppet function that accepts a particular set of arguments and whose return value obeys a particular format. The function can do whatever is necessary to locate its data.

## Convert a version 3 `hiera.yaml` to version 5

Hiera 5 supports three versions of the `hiera.yaml` file: version 3, version 4, and version 5. If you've been using Hiera 3, your existing configuration is a version 3 `hiera.yaml` file at the global layer.

There are two migration tasks that involve translating a version 3 config to a version 5:

- Creating new v5 `hiera.yaml` files for environments.
- Updating your global configuration to support Hiera 5 backends.

These are essentially the same process, although the global hierarchy has a few special capabilities.

Consider this example `hiera.yaml` version 3 file:

```
:backends:
  - mongodb
  - eyaml
  - yaml
:yaml:
  :datadir: "/etc/puppetlabs/code/environments/%{environment}/hieradata"
:mongodb:
  :connections:
    :dbname: hdata
    :collection: config
    :host: localhost
:eyaml:
  :datadir: "/etc/puppetlabs/code/environments/%{environment}/hieradata"
  :pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
  :pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem
:hierarchy:
  - "nodes/%{trusted.certname}"
  - "location/%{facts.whereami}/%{facts.group}"
  - "groups/%{facts.group}"
  - "os/%{facts.os.family}"
  - "common"
:logger: console
:merge_behavior: native
:deep_merge_options: {}
```

To convert this version 3 file to version 5:

**1.** Use strings instead of symbols for keys.

   Hiera 3 required you to use Ruby symbols as keys. Symbols are short strings that start with a colon, for example, `:hierarchy`. The version 5 config format lets you use regular strings as keys, although symbols won't (yet) cause errors. You can remove the leading colons on keys.

**2.** Remove settings that aren't used anymore. In this example, remove everything except the `:hierarchy` setting:

   a) Delete the following settings completely, which are no longer needed:

   - `:logger`
   - `:merge_behavior`
   - `:deep_merge_options`

   For information on how Hiera 5 supports deep hash merging, see Merging data from multiple sources.

   b) Delete the following settings, but paste them into a temporary file for later reference:

   - `:backends`
   - Any backend-specific setting sections, like `:yaml` or `:mongodb`

**3.** Add a `version` key, with a value of `5`:

```
version: 5
hierarchy:
  # ...
```

4. Set a default backend and data directory.

   If you use one backend for the majority of your data, for example YAML or JSON, set a `defaults` key, with values for `datadir` and one of the backend keys.

   The names of the backends have changed for Hiera 5, and the `backend` setting itself has been split into three settings:

| Hiera 3 backend | Hiera 5 backend setting |
|---|---|
| `yaml` | `data_hash: yaml_data` |
| `json` | `data_hash: json_data` |
| `eyaml` | `lookup_key: eyaml_lookup_key` |

5. Translate the hierarchy.

   The version 5 and version 3 hierarchies work differently:

   - In version 3, hierarchy levels don't have a backend assigned to them, and Hiera loops through the entire hierarchy for each backend.
   - In version 5, each hierarchy level has one designated backend, as well as its own independent configuration for that backend.

   Consult the previous values for the `:backends` key and any backend-specific settings.

   In the example above, we used `yaml`, `eyaml`, and `mongodb` backends. Your business only uses Mongo for per-node data, and uses eyaml for per-group data. The rest of the hierarchy is irrelevant to these backends. You need one Mongo level and one eyaml level, but still want all five levels in YAML. This means Hiera consults multiple backends for per-node and per-group data. You want the YAML version of per-node data to be authoritative, so put it before the Mongo version. The eyaml data does not overlap with the unencrypted per-group data, so it doesn't matter where you put it. Put it before the YAML levels. When you translate your hierarchy, you have to make the same kinds of investigations and decisions.
6. Remove hierarchy levels that use `calling_module`, `calling_class`, and `calling_class_path`, which were allowed pseudo-variables in Hiera 3. Anything you were doing with these variables is better accomplished by using the module data layer, or by using the glob pattern (if the reason for using them was to enable splitting up data into multiple files, and not knowing in advance what they names of those would be)

   `Hiera.yaml` version 5 does not support these. Remove hierarchy levels that interpolate them.

   Anything you were doing with these variables is better accomplished by using the module data layer, or by using the glob pattern (if the reason for using them was to enable splitting up data into multiple files, and not knowing in advance what they names of those would be)
7. Translate built-in backends to the version 5 config, where the hierarchy is written as an array of hashes. For hierarchy levels that use the built-in backends, for example YAML and JSON, use the `data_hash` key to set the backend. See Configuring a hierarchy level in the `hiera.yaml` v5 reference for more information.

   Set the following keys:

   - `name` - A human-readable name.
   - `path` or `paths` - The path you used in your version 3 `hiera.yaml` hierarchy, but with a file extension appended.
   - `data_hash` - The backend to use `yaml_data` for YAML, `json_data` for JSON.
   - `datadir` - The data directory. In version 5, it's relative to the `hiera.yaml` file's directory.

   If you have set default values for `data_hash` and `datadir`, you can omit them.

```
version: 5
defaults:
  datadir: data
  data_hash: yaml_data
hierarchy:
```

```
  - name: "Per-node data (yaml version)"
    path: "nodes/%{trusted.certname}.yaml" # Add file extension.
    # Omitting datadir and data_hash to use defaults.

  - name: "Other YAML hierarchy levels"
    paths: # Can specify an array of paths instead of one.
      - "location/%{facts.whereami}/%{facts.group}.yaml"
      - "groups/%{facts.group}.yaml"
      - "os/%{facts.os.family}.yaml"
      - "common.yaml"
```

**8.** Translate `hiera-eyaml` backends, which work in a similar way to the other built-in backends.

The differences are:

- The `hiera-eyaml` gem has to be installed, and you need a different backend setting. Instead of `data_hash: yaml`, use `lookup_key: eyaml_lookup_key`. Each hierarchy level needs an `options` key with paths to the public and private keys. You cannot set a global default for this.

```
- name: "Per-group secrets"
   path: "groups/%{facts.group}.eyaml"
   lookup_key: eyaml_lookup_key
   options:
     pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/
private_key.pkcs7.pem
     pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/
public_key.pkcs7.pem
```

**9.** Translate custom Hiera 3 backends.

Check to see if the backend's author has published a Hiera 5 update for it. If so, use that; see its documentation for details on how to configure hierarchy levels for it.

If there is no update, use the version 3 backend in a version 5 hierarchy at the global layer — it does not work in the environment layer. Find a Hiera 5 compatible replacement, or write Hiera 5 backends yourself.

For details on how to configure a legacy backend, see Configuring a hierarchy level (legacy Hiera 3 backends) in the `hiera.yaml` (version 5) reference.

When configuring a legacy backend, use the previous value for its backend-specific settings. In the example, the version 3 config had the following settings for MongoDB:

```
:mongodb:
  :connections:
    :dbname: hdata
    :collection: config
    :host: localhost
```

So, write the following for a per-node MongoDB hierarchy level:

```
- name: "Per-node data (MongoDB version)"
   path: "nodes/%{trusted.certname}"      # No file extension
   hiera3_backend: mongodb
   options:    # Use old backend-specific options, changing keys to plain
 strings
     connections:
       dbname: hdata
       collection: config
       host: localhost
```

After following these steps, you've translated the example configuration into the following v5 config:

```
version: 5
defaults:
```

```
  datadir: data
  data_hash: yaml_data
hierarchy:
  - name: "Per-node data (yaml version)"
    path: "nodes/%{trusted.certname}.yaml" # Add file extension
    # Omitting datadir and data_hash to use defaults.

  - name: "Per-node data (MongoDB version)"
    path: "nodes/%{trusted.certname}"      # No file extension
    hiera3_backend: mongodb
    options:    # Use old backend-specific options, changing keys to plain
 strings
      connections:
        dbname: hdata
        collection: config
        host: localhost

  - name: "Per-group secrets"
    path: "groups/%{facts.group}.eyaml"
    lookup_key: eyaml_lookup_key
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
      pkcs7_public_key:  /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem

  - name: "Other YAML hierarchy levels"
    paths: # Can specify an array of paths instead of a single one.
      - "location/%{facts.whereami}/%{facts.group}.yaml"
      - "groups/%{facts.group}.yaml"
      - "os/%{facts.os.family}.yaml"
      - "common.yaml"
```

**Related information**

Hiera configuration layers on page 398
Hiera uses three independent layers of configuration. Each layer has its own hierarchy, and they’re linked into one super-hierarchy before doing a lookup.

Custom backends overview on page 423
A backend is a custom Puppet function that accepts a particular set of arguments and whose return value obeys a particular format. The function can do whatever is necessary to locate its data.

Configuring a hierarchy level: general format on page 409
Hiera supports custom backends.

Configuring a hierarchy level: hiera-eyaml on page 407
Hiera 5 ( Puppet 4.9.3 and later) includes a native interface for the Hiera eyaml extension, which keeps data encrypted on disk but lets Puppet read it during catalog compilation.

## Convert an experimental (version 4) `hiera.yaml` to version 5

If you used the experimental version of Puppet lookup ( Hiera 5's predecessor), you might have some version 4 `hiera.yaml` files in your environments and modules. Hiera 5 can use these, but you need to convert them, especially if you want to use any backends other than YAML or JSON. Version 4 and version 5 formats are similar.

Consider this example of a version 4 `hiera.yaml` file:

```
# /etc/puppetlabs/code/environments/production/hiera.yaml
---
version: 4
datadir: data
hierarchy:
  - name: "Nodes"
    backend: yaml
    path: "nodes/%{trusted.certname}"
```

```
  - name: "Exported JSON nodes"
    backend: json
    paths:
      - "nodes/%{trusted.certname}"
      - "insecure_nodes/%{facts.networking.fqdn}"

  - name: "virtual/%{facts.virtual}"
    backend: yaml

  - name: "common"
    backend: yaml
```

To convert to version 5, make the following changes:

1. Change the value of the `version` key to `5`.
2. Add a file extension to every file path — use `"common.yaml"`, not `"common"`.
3. If any hierarchy levels are missing a path, add one. In version 5, path no longer defaults to the value of name
4. If there is a top-level `datadir` key, change it to a `defaults` key. Set a default backend. For example:

   ```
   defaults:
     datadir: data
     data_hash: yaml_data
   ```

5. In each hierarchy level, delete the `backend` key and replace it with a `data_hash` key. (If you set a default backend in the defaults key, you can omit it here.)

   | v4 backend | v5 equivalent |
   | --- | --- |
   | `backend: yaml` | `data_hash: yaml_data` |
   | `backend: json` | `data_hash: json_data` |

6. Delete the `environment_data_provider` and `data_provider` settings, which enabled Puppet lookup for an environment or module.

   You'll find these settings in the following locations:

   - `environment_data_provider` in `puppet.conf`.
   - `environment_data_provider` in `environment.conf`.
   - `data_provider` in a module's `metadata.json`.

After being converted to version 5, the example looks like this:

```
# /etc/puppetlabs/code/environments/production/hiera.yaml
---
version: 5
defaults:
  datadir: data          # Datadir has moved into `defaults`.
  data_hash: yaml_data   # Default backend: New feature in v5.
hierarchy:
  - name: "Nodes"        # Can omit `backend` if using the default.
    path: "nodes/%{trusted.certname}.yaml"   # Add file extension!

  - name: "Exported JSON nodes"
    data_hash: json_data        # Specifying a non-default backend.
    paths:
      - "nodes/%{trusted.certname}.json"
      - "insecure_nodes/%{facts.networking.fqdn}.json"

  - name: "Virtualization platform"
    path: "virtual/%{facts.virtual}.yaml"   # Name and path are now
 separated.
```

```
- name: "common"
  path: "common.yaml"
```

For full syntax details, see the `hiera.yaml` version 5 reference.

**Related information**

Config file syntax on page 403
The `hiera.yaml` file is a YAML file, containing a hash with up to four top-level keys.

Custom backends overview on page 423
A backend is a custom Puppet function that accepts a particular set of arguments and whose return value obeys a particular format. The function can do whatever is necessary to locate its data.

## Convert experimental data provider functions to a Hiera 5 `data_hash` backend

Puppet lookup had experimental custom backend support, where you could set `data_provider = function` and create a function with a name that returned a hash. If you used that, you can convert your function to a Hiera 5 `data_hash` backend.

1. Your original function took no arguments. Change its signature to accept two arguments: a `Hash` and a `Puppet::LookupContext` object. You do not have to do anything with these - just add them. For more information, see the documentation for data hash backends.
2. Delete the `data_provider` setting, which enabled Puppet lookup for a module. You can find this setting in a module's `metadata.json`.
3. Create a version 5 `hiera.yaml` file for the affected environment or module, and add a hierarchy level as follows:

   ```
   - name: <ARBITRARY NAME>
     data_hash: <NAME OF YOUR FUNCTION>
   ```

   It does not need a `path`, `datadir`, or any other options.

## Updated classic Hiera function calls

The `hiera`, `hiera_array`, `hiera_hash`, and `hiera_include` functions are all deprecated. The `lookup` function is a complete replacement for all of these.

| Hiera function | Equivalent `lookup` call |
|---|---|
| `hiera('secure_server')` | `lookup('secure_server')` |
| `hiera_array('ntp::servers')` | `lookup('ntp::servers', {merge => unique})` |
| `hiera_hash('users')` | `lookup('users', {merge => hash})` or `lookup('users', {merge => deep})` |
| `hiera_include('classes')` | `lookup('classes', {merge => unique}).include` |

To prepare for deprecations in future Puppet versions, it's best to revise your Puppet modules to replace the `hiera_*` functions with `lookup`. However, you can adopt all of Hiera 5's new features without updating these function calls. While you're revising, consider refactoring code to use automatic class parameter lookup instead of manual lookup calls. Because automatic lookups can now do unique and hash merges, the use of manual lookup in the form of `hiera_array` and `hiera_hash` are not as important as they used to be. Instead of changing those manual Hiera calls to be calls to the `lookup` function, use Automatic Parameter Lookup (API).

**Related information**

The Puppet lookup function on page 419

The `lookup` function uses Hiera to retrieve a value for a given key.

Merge behaviors on page 410
There are four merge behaviors to choose from: `first`, `unique`, `hash`, and `deep`.

## Adding Hiera data to a module

Modules need default values for their class parameters. Before, the preferred way to do this was the "params.pp" pattern. With Hiera 5, you can use the "data in modules" approach instead. The following example shows how to replace `params.pp` with the new approach.

**Note:** The `params.pp` pattern is still valid, and the features it relies on remain in Puppet. But if you want to use Hiera data instead, you now have that option.

### Module data with the `params.pp` pattern

The `params.pp` pattern takes advantage of the Puppet class inheritance behavior.

One class in your module does nothing but set variables for the other classes. This class is called `<MODULE>::params`. This class uses Puppet code to construct values; it uses conditional logic based on the target operating system. The rest of the classes in the module inherit from the params class. In their parameter lists, you can use the params class's variables as default values.

When using `params.pp` pattern, the values set in the `params.pp` defined class cannot be used in lookup merges and Automatic Parameter Lookup (APL) - when using this pattern these are only used for defaults when there are no values found in Hiera.

An example params class:

```
# ntp/manifests/params.pp
class ntp::params {
  $autoupdate = false,
  $default_service_name = 'ntpd',

  case $facts['os']['family'] {
    'AIX': {
      $service_name = 'xntpd'
    }
    'Debian': {
      $service_name = 'ntp'
    }
    'RedHat': {
      $service_name = $default_service_name
    }
  }
}
```

A class that inherits from the params class and uses it to set default parameter values:

```
class ntp (
  $autoupdate   = $ntp::params::autoupdate,
  $service_name = $ntp::params::service_name,
) inherits ntp::params {
 ...
}
```

### Module data with a one-off custom Hiera backend

With Hiera 5's custom backend system, you can convert and existing params class to a hash-based Hiera backend.

To create a Hiera backend, create a function written in the Puppet language that returns a hash.

Using the `params` class as a starting point:

```
# ntp/functions/params.pp
```

```
function ntp::params(
  Hash                  $options, # We ignore both of these arguments, but
  Puppet::LookupContext $context, # the function still needs to accept them.
) {
  $base_params = {
    'ntp::autoupdate'   => false,
      # Keys have to start with the module's namespace, which in this case
 is `ntp::`.
    'ntp::service_name' => 'ntpd',
      # Use key names that work with automatic class parameter lookup. This
      # key corresponds to the `ntp` class's `$service_name` parameter.
  }

  $os_params = case $facts['os']['family'] {
    'AIX': {
      { 'ntp::service_name' => 'xntpd' }
    }
    'Debian': {
      { 'ntp::service_name' => 'ntp' }
    }
    default: {
      {}
    }
  }

  # Merge the hashes, overriding the service name if this platform uses a
 non-standard one:
  $base_params + $os_params
}
```

**Note:** The hash merge operator (+) is useful in these functions.

After you have a function, tell Hiera to use it by adding it to the module layer `hiera.yaml`. A backend like this one doesn't require `path`, `datadir`, or `options` keys. You have a choice of adding it to the `default_hierarch` if you want the exact same behaviour as with the earlier `params.pp` pattern, and use the regular `hierarchy` if you want the values to be merged with values of higher priority when a merging lookup is specified. You can split up the key-values so that some are in the `hierarchy`, and some in the `default_hierarchy`, depending on whether it makes sense to merge a value or not.

Here we add it to the regular hierarchy:

```
# ntp/hiera.yaml
---
version: 5
hierarchy:
  - name: "NTP class parameter defaults"
    data_hash: "ntp::params"
  # We only need one hierarchy level, because one function provides all the
 data.
```

With Hiera-based defaults, you can simplify your module's main classes:

- They do not need to inherit from any other class.
- You do not need to explicitly set a default value with the = operator.
- Instead APL comes into effect for each parameter without a given value. In the example, the function `ntp::params` is called to get the default params, and those can then be either overridden or merged, just as with all values in Hiera.

  ```
  # ntp/manifests/init.pp
  class ntp (
    # default values are in ntp/functions/params.pp
    $autoupdate,
  ```

```
  $service_name,
) {
 ...
}
```

### Module data with YAML data files

You can also manage your module's default data with basic Hiera YAML files,

Set up a hierarchy in your module layer `hiera.yaml` file:

```
# ntp/hiera.yaml
---
version: 5
defaults:
  datadir: data
  data_hash: yaml_data
hierarchy:
  - name: "OS family"
    path: "os/%{facts.os.family}.yaml"

  - name: "common"
    path: "common.yaml"
```

Then, put the necessary data files in the data directory:

```
# ntp/data/common.yaml
---
ntp::autoupdate: false
ntp::service_name: ntpd

# ntp/data/os/AIX.yaml
---
ntp::service_name: xntpd

# ntp/data/os/Debian.yaml
ntp::service_name: ntp
```

You can also use any other Hiera backend to provide your module's data. If you want to use a custom backend that is distributed as a separate module, you can mark that module as a dependency.

For more information, see class inheritance, conditional logic, write functions in the Puppet language, hash merge operator.

### Related information

The Puppet lookup function on page 419
The `lookup` function uses Hiera to retrieve a value for a given key.

Hiera configuration layers on page 398
Hiera uses three independent layers of configuration. Each layer has its own hierarchy, and they're linked into one super-hierarchy before doing a lookup.

# Facter

Facter is Puppet's cross-platform system profiling library. It discovers and reports per-node facts, which are available in your Puppet manifests as variables.

- Facter release notes on page 443
These are the new features, resolved issues, and deprecations in this version of Facter.
- Core facts

- Custom facts on page 444

A typical fact in Facter is a collection of several elements, and is written either as a simple value ("flat" fact) or as structured data ("structured" fact).

- Custom facts walkthrough on page 448

You can add custom facts by writing snippets of Ruby code on the Puppet master. Puppet then uses plug-ins in modules to distribute the facts to the client.

- External facts on page 453

External facts provide a way to use arbitrary executables or scripts as facts, or set facts statically with structured data. With this information, you can write a custom fact in Perl, C, or a one-line text file.

- Configuring Facter with facter.conf on page 457

The `facter.conf` file is a configuration file that allows you to cache and block fact groups, and manage how Facter interacts with your system. There are three sections: `facts`, `global`, and `cli`. All sections are optional and can be listed in any order within the file.

# Facter release notes

These are the new features, resolved issues, and deprecations in this version of Facter.

## Facter 3.14.2

Released 23 July 2019 and shipped with Puppet Platform 6.7.0.

### Resolved issues

- Prior to this release, Facter returned warnings if `ip route show` output was not in a key-value format. Because this format does not apply to all configurations, Facter no longer returns warnings about it. FACT-1916
- Previously, the `mountpoint` fact showed only temporary file systems and physical mounts. Now Facter returns mount points for all mounts on the system. FACT-1910

## Facter 3.14.1

Released 23 July 2019 and shipped with Puppet Platform 6.7.0.

### New features

- Facter now report disk serial numbers on Linux and FreeBSD FACT-1929
- This release adds a primary network interface check for FreeBSD. FACT-1926
- Previously, the `mountpoint` fact showed only temporary file systems and physical mounts. Now Facter returns mount points for all mounts on the system. FACT-1910
- This release adds support for Virtuzzo Linux facts. FACT-1888

## Facter 3.14.0

Released 23 July 2019 and shipped with Puppet Platform 6.7.2

### New features

In this release, Facter adds new facts for Windows version 10/2016+:

- `ReleaseID`: The four-digit Windows build version, in the form `YYMM`. On Windows10-1511-x86_64, the release ID is not displayed, as is not present in the registry.
- `InstallationType`: Differentiates Server, Server Core, Client (Desktop): `Server|Server Core|Client`.
- `EditionID`: Server or Desktop Edition variant: `ServerStandard|Professional|Enterprise`.
- `ProductName`: Textual Product Name.

# Custom facts

A typical fact in Facter is a collection of several elements, and is written either as a simple value ("flat" fact) or as structured data ("structured" fact).

This page is an example-driven tour of those elements, and is a quick primer or reference for people writing custom facts. You need some familiarity with Ruby to understand most of these examples. For a gentler introduction, check out the Custom facts walkthrough.

It's important to distinguish between facts and resolutions. A fact is a piece of information about a given node, while a resolution is a way of obtaining that information from the system. Every fact must have at least one resolution, and facts that can run on different operating systems could have different resolutions for each OS.

Although facts and resolutions are conceptually very different, the line can get a bit blurry at times, because declaring a second (or more) resolution for a fact looks just like declaring a completely new fact, with the same name as an existing fact.

For information on how to add custom facts to modules, see Plug-ins in modules.

## Writing facts with simple resolutions

Most facts are resolved all at the same time, without any need to merge data from different sources. In that case, the resolution is simple. Both flat and structured facts can have simple resolutions.

### Example: Minimal fact that relies on a single shell command

```
Facter.add(:rubypath) do
  setcode 'which ruby'
end
```

### Example: Different resolutions for different operating systems

```
Facter.add(:rubypath) do
  setcode 'which ruby'
end

Facter.add(:rubypath) do
  confine :osfamily => "Windows"
  # Windows uses 'where' instead of 'which'
  setcode 'where ruby'
end
```

### Example: Slightly more complex fact, confined to Linux with a block

```
Facter.add(:jruby_installed) do
  confine :kernel do |value|
    value == "Linux"
  end

  setcode do
    # If jruby is present, return true. Otherwise, return false.
    Facter::Core::Execution.which('jruby') != nil
  end
end
```

### Main components of simple resolutions

Simple facts are typically made up of the following parts:

1. A call to `Facter.add(:fact_name)`:
   - This introduces a new fact *or* a new resolution for an existing fact with the same name.
   - The name can be either a symbol or a string.
   - The rest of the fact is wrapped in the `add` call's `do ... end` block.
2. Zero or more `confine` statements:
   - Determine whether the resolution is suitable (and therefore is evaluated).
   - Can either match against the value of another fact or evaluate a Ruby block.
   - If given a symbol or string representing a fact name, a block is required and the block receives the fact's value as an argument.
   - If given a hash, the keys are expected to be fact names. The values of the hash are either the expected fact values or an array of values to compare against.
   - If given a block, the confine is suitable if the block returns a value other than `nil` or `false`.
3. An optional `has_weight` statement:
   - When multiple resolutions are available for a fact, resolutions are evaluated from highest weight value to lowest.
   - Must be an integer greater than 0.
   - Defaults to the number of `confine` statements for the resolution.
4. A `setcode` statement that determines the value of the fact:
   - Can take either a string or a block.
   - If given a string, Facter executes it as a shell command. If the command succeeds, the output of the command is the value of the fact. If the command fails, the next suitable resolution is evaluated.
   - If given a block, the block's return value is the value of the fact unless the block returns `nil`. If `nil` is returned, the next suitable resolution is evalutated.
   - Can execute shell commands within a `setcode` block, using the `Facter::Core::Execution.exec` function.
   - If multiple `setcode` statements are evaluated for a single resolution, only the last `setcode` block is used.

## Writing structured facts

Structured facts can take the form of hashes or arrays.

You don't have to do anything special to mark the fact as structured — if your fact returns a hash or array, Facter recognizes it as a structured fact. Structured facts can have simple or aggregate resolutions.

### Example: Returning an array of network interfaces

```
Facter.add(:interfaces_array) do
  setcode do
   interfaces = Facter.value(:interfaces)
   # the 'interfaces' fact returns a single comma-delimited string, such as
 "lo0,eth0,eth1"
   # this splits the value into an array of interface names
   interfaces.split(',')
  end
end
```

### Example: Returning a hash of network interfaces to IP addresses

```
Facter.add(:interfaces_hash) do
  setcode do
    interfaces_hash = {}

    Facter.value(:interfaces_array).each do |interface|
      ipaddress = Facter.value("ipaddress_#{interface}")
```

```
      if ipaddress
        interfaces_hash[interface] = ipaddress
      end
    end

    interfaces_hash
  end
end
```

## Writing facts with aggregate resolutions

Aggregate resolutions allow you to split up the resolution of a fact into separate chunks.

By default, Facter merges hashes with hashes or arrays with arrays, resulting in a structured fact, but you can also aggregate the chunks into a flat fact using concatenation, addition, or any other function that you can express in Ruby code.

### Main components of aggregate resolutions

Aggregate resolutions have two key differences compared to simple resolutions: the presence of `chunk` statements and the lack of a `setcode` statement. The `aggregate` block is optional, and without it Facter merges hashes with hashes or arrays with arrays.

1. A call to `Facter.add(:fact_name, :type => :aggregate)`:
   - Introduces a new fact or a new resolution for an existing fact with the same name.
   - The name can be either a symbol or a string.
   - The `:type => :aggregate` parameter is required for aggregate resolutions.
   - The rest of the fact is wrapped in the `add` call's `do ... end` block.
2. Zero or more `confine` statements:
   - Determine whether the resolution is suitable and (therefore is evaluated).
   - They can either match against the value of another fact or evaluate a Ruby block.
   - If given a symbol or string representing a fact name, a block is required and the block receives the fact's value as an argument.
   - If given a hash, the keys are expected to be fact names. The values of the hash are either the expected fact values or an array of values to compare against.
   - If given a block, the confine is suitable if the block returns a value other than `nil` or `false`.
3. An optional `has_weight` statement:
   - Evaluates multiple resolutions for a fact from highest weight value to lowest.
   - Must be an integer greater than 0.
   - Defaults to the number of `confine` statements for the resolution.
4. One or more calls to `chunk`, each containing:
   - A name (as the argument to `chunk`).
   - A block of code, which is responsible for resolving the chunk to a value. The block's return value is the value of the chunk; it can be any type, but is typically a hash or array.
5. An optional `aggregate` block:
   - If absent, Facter automatically merges hashes with hashes or arrays with arrays.
   - To merge the chunks in any other way, you need to make a call to `aggregate`, which takes a block of code.
   - The block is passed one argument (`chunks`, in the example), which is a hash of chunk name to chunk value for all the chunks in the resolution.

### Example: Building a structured fact progressively

This example builds a new fact, `networking_primary_sha`, by progressively merging two chunks. One chunk encodes each networking interface's MAC address as an encoded base64 value, and the other determines if each interface is the system's primary interface.

```
require 'digest'
require 'base64'

Facter.add(:networking_primary_sha, :type => :aggregate) do

  chunk(:sha256) do
    interfaces = {}

    Facter.value(:networking)['interfaces'].each do |interface, values|
      if values['mac']
        hash = Digest::SHA256.digest(values['mac'])
        encoded = Base64.encode64(hash)
        interfaces[interface] = {:mac_sha256 => encoded.strip}
      end
    end

    interfaces
  end

  chunk(:primary?) do
    interfaces = {}

    Facter.value(:networking)['interfaces'].each do |interface, values|
      interfaces[interface] = {:primary? => (interface ==
 Facter.value(:networking)['primary'])}
    end

    interfaces
  end
  # Facter merges the return values for the two chunks
  # automatically, so there's no aggregate statement.
end
```

The fact's output is organized by network interface into hashes, each containing the two chunks:

```
{
  bridge0 => {
    mac_sha256 => "bfgEFV7m1V04HYU6UqzoNoVmnPIEKWRSUOU650j0Wkk=",
    primary?   => false
  },
  en0 => {
    mac_sha256 => "6Fd3Ws2z+aIl8vNmClCbzxiO2TddyFBChMlIU+QB28c=",
    primary?   => true
  },
  ...
}
```

### Example: Building a flat fact progressively with addition

```
Facter.add(:total_free_memory_mb, :type => :aggregate) do
  chunk(:physical_memory) do
    Facter.value(:memoryfree_mb)
  end

  chunk(:virtual_memory) do
    Facter.value(:swapfree_mb)
```

```
  end

  aggregate do |chunks|
    # The return value for this block determines the value of the fact.
    sum = 0
    chunks.each_value do |i|
      sum += i
    end

    sum
  end
end
```

## Custom facts walkthrough

You can add custom facts by writing snippets of Ruby code on the Puppet master. Puppet then uses plug-ins in modules to distribute the facts to the client.

For information on how to add custom facts to modules, see Plug-ins in modules.

#### Related information

External facts on page 453
External facts provide a way to use arbitrary executables or scripts as facts, or set facts statically with structured data. With this information, you can write a custom fact in Perl, C, or a one-line text file.

### Adding custom facts to Facter

Sometimes you need to be able to write conditional expressions based on site-specific data that just isn't available via Facter, or perhaps you'd like to include it in a template.

Because you can't include arbitrary Ruby code in your manifests, the best solution is to add a new fact to Facter. These additional facts can then be distributed to Puppet clients and are available for use in manifests and templates, just like any other fact is.

**Note:** Facter 3.0 removed the Ruby implementations of some features and replaced them with a custom facts API. Any custom fact that requires one of the Ruby files previously stored in `lib/facter/util` fails with an error.

#### Structured and flat facts

A typical fact extracts a piece of information about a system and returns it as either as a simple value ("flat" fact) or data organized as a hash or array ("structured" fact). There are several types of facts classified by how they collect information, including:

- Core facts, which are built into Facter and are common to almost all systems.
- Custom facts, which run Ruby code to produce a value.
- External facts, which return values from pre-defined static data, or the result of an executable script or program.

All fact types can produce flat or structured values.

#### Related information

Facter release notes on page 443
These are the new features, resolved issues, and deprecations in this version of Facter.

External facts on page 453
External facts provide a way to use arbitrary executables or scripts as facts, or set facts statically with structured data. With this information, you can write a custom fact in Perl, C, or a one-line text file.

### Loading custom facts

Facter offers multiple methods of loading facts.

These include:

- `$LOAD\_PATH`, or the Ruby library load path.
- The `--custom-dir` command line option.
- The environment variable `FACTERLIB`.

You can use these methods to do things like test files locally before distributing them, or you can arrange to have a specific set of facts available on certain machines.

### Using the Ruby load path

Facter searches all directories in the Ruby `$LOAD_PATH` variable for subdirectories named Facter, and loads all Ruby files in those directories. If you had a directory in your `$LOAD_PATH` like `~/lib/ruby`, set up like this:

```
#~/lib/ruby
└── facter
    ├── rackspace.rb
    ├── system_load.rb
    └── users.rb
```

Facter loads `facter/system_load.rb`, `facter/users.rb`, and `facter/rackspace.rb`.

### Using the `--custom-dir` command line option

Facter can take multiple `--custom-dir` options on the command line that specifies a single directory to search for custom facts. Facter attempts to load all Ruby files in the specified directories. This allows you to do something like this:

```
$ ls my_facts
system_load.rb
$ ls my_other_facts
users.rb
$ facter --custom-dir=./my_facts --custom-dir=./my_other_facts system_load
 users
system_load => 0.25
users => thomas,pat
```

### Using the `FACTERLIB` environment variable

Facter also checks the environment variable `FACTERLIB` for a delimited (semicolon for Windows and colon for all other platforms) set of directories, and tries to load all Ruby files in those directories. This allows you to do something like this:

```
$ ls my_facts
system_load.rb
$ ls my_other_facts
users.rb
$ export FACTERLIB="./my_facts:./my_other_facts"
$ facter system_load users
system_load => 0.25
users => thomas,pat
```

## Two parts of every fact

Most facts have at least two elements.

1. A call to `Facter.add('fact_name')`, which determines the name of the fact.
2. A `setcode` statement for simple resolutions, which is evaluated to determine the fact's value.

Facts can get a lot more complicated than that, but those two together are the most common implementation of a custom fact.

## Executing shell commands in facts

Puppet gets information about a system from Facter, and the most common way for Facter to get that information is by executing shell commands.

You can then parse and manipulate the output from those commands using standard Ruby code. The Facter API gives you a few ways to execute shell commands:

- To run a command and use the output verbatim, as your fact's value, you can pass the command into `setcode` directly. For example: `setcode 'uname --hardware-platform'`
- If your fact is more complicated than that, you can call `Facter::Core::Execution.execute('uname --hardware-platform')` from within the `setcode do ... end` block. Whatever the `setcode` statement returns is used as the fact's value.
- Your shell command is also a Ruby string, so you need to escape special characters if you want to pass them through.

**Note:** Not everything that works in the terminal works in a fact. You can use the pipe (`|`) and similar operators as you normally would, but Bash-specific syntax like `if` statements do not work. The best way to handle this limitation is to write your conditional logic in Ruby.

### Example

To get the output of `uname --hardware-platform` to single out a specific type of workstation, you create a custom fact.

1. Start by giving the fact a name, in this case, `hardware_platform`.
2. Create the fact in a file called `hardware_platform.rb` on the Puppet master server:

```
# hardware_platform.rb

Facter.add('hardware_platform') do
  setcode do
    Facter::Core::Execution.execute('/bin/uname --hardware-platform')
  end
end
```

3. Use the instructions in the Plug-ins in modules docs to copy the new fact to a module and distribute it. During your next Puppet run, the value of the new fact is available to use in your manifests and templates.

## Using other facts

You can write a fact that uses other facts by accessing `Facter.value(:somefact)`. If the fact fails to resolve or is not present, Facter returns `nil`.

For example:

```
Facter.add(:osfamily) do
  setcode do
    distid = Facter.value(:lsbdistid)
    case distid
    when /RedHatEnterprise|CentOS|Fedora/
      'redhat'
    when 'ubuntu'
      'debian'
    else
      distid
    end
  end
end
```

## Configuring facts

Facts have properties that you can use to customize how they are evaluated.

### Confining facts

One of the more commonly used properties is the `confine` statement, which restricts the fact to run only on systems that match another given fact.

For example:

```
Facter.add(:powerstates) do
  confine :kernel => 'Linux'
  setcode do
    Facter::Core::Execution.execute('cat /sys/power/states')
  end
end
```

This fact uses `sysfs` on Linux to get a list of the power states that are available on the given system. Because this is available only on Linux systems, we use the `confine` statement to ensure that this fact isn't needlessly run on systems that don't support this type of enumeration.

To confine structured facts like `['os']['family']`, you can use `Facter.value`:

```
confine Facter.value(:os)['family'] => 'RedHat'
```

You can also use a Ruby block:

```
confine :os do |os|
  os['family'] == 'RedHat'
end
```

### Fact precedence

A single fact can have multiple *resolutions*, each of which is a different way of determining the value of the fact. It's common to have different resolutions for different operating systems, for example. To add a new resolution to a fact, you add the fact again with a different `setcode` statement.

When a fact has more than one resolution, the first resolution that returns a value other than `nil` sets the fact's value. The way that Facter decides the issue of resolution precedence is the weight property. After Facter rules out any resolutions that are excluded because of `confine` statements, the resolution with the highest weight is evaluated first. If that resolution returns `nil`, Facter moves on to the next resolution (by descending weight) until it gets a value for the fact.

By default, the weight of a resolution is the number of `confine` statements it has, so that more specific resolutions take priority over less specific resolutions. External facts have a weight of 1000 — to override them, set a weight above 1000.

```
# Check to see if this server has been marked as a postgres server
Facter.add(:role) do
  has_weight 100
  setcode do
    if File.exist? '/etc/postgres_server'
      'postgres_server'
    end
  end
end

# Guess if this is a server by the presence of the pg_create binary
Facter.add(:role) do
  has_weight 50
```

```
  setcode do
    if File.exist? '/usr/sbin/pg_create'
      'postgres_server'
    end
  end
end

# If this server doesn't look like a server, it must be a desktop
Facter.add(:role) do
  setcode do
    'desktop'
  end
end
```

#### Execution timeouts

Although this version of Facter does not support overall timeouts on resolutions, you can pass a timeout to `Facter::Core::Execution#execute`:

```
Facter.add(:sleep) do
  setcode do
    begin
      Facter::Core::Execution.execute('sleep 10', options = {:timeout => 5})
      'did not timeout!'
    rescue Facter::Core::Execution::ExecutionFailure
      Facter.warn("Sleep fact timed out!")
    end
  end
end
```

When Facter runs as standalone, using `Facter.warn` ensures that the message is printed to `STDERR`. When Facter is called as part of a catalog application, using `Facter.warn` prints the message to Puppet's log. If an exception is not caught, Facter automatically logs it as an error.

### Structured facts

Structured facts take the form of either a hash or an array.

To create a structured fact, return a hash or an array from the `setcode` statement.

You can see some relevant examples in the Writing structured facts section of the Custom facts overview.

#### Related information

Custom facts on page 444
A typical fact in Facter is a collection of several elements, and is written either as a simple value ("flat" fact) or as structured data ("structured" fact).

### Aggregate resolutions

If your fact combines the output of multiple commands, use aggregate resolutions. An aggregate resolution is split into chunks, each one responsible for resolving one piece of the fact. After all of the chunks have been resolved separately, they're combined into a single flat or structured fact and returned.

Aggregate resolutions have several key differences compared to simple resolutions, beginning with the fact declaration. To introduce an aggregate resolution, add the `:type => :aggregate` parameter:

```
Facter.add(:fact_name, :type => :aggregate) do
    #chunks go here
    #aggregate block goes here
end
```

Each step in the resolution then gets its own named `chunk` statement:

```
chunk(:one) do
    'Chunk one returns this. '
end

chunk(:two) do
    'Chunk two returns this.'
end
```

Aggregate resolutions never have a `setcode` statement. Instead, they have an optional `aggregate` block that combines the chunks. Whatever value the `aggregate` block returns is the fact's value. Here's an example that just combines the strings from the two chunks above:

```
aggregate do |chunks|
  result = ''

  chunks.each_value do |str|
    result += str
  end

  # Result: "Chunk one returns this. Chunk two returns this."
  result
end
```

If the `chunk` blocks all return arrays or hashes, you can omit the `aggregate` block. If you do, Facter merges all of your data into one array or hash and uses that as the fact's value.

For more examples of aggregate resolutions, see the Aggregate resolutions section of the Custom facts overview page.

#### Related information

Custom facts on page 444
A typical fact in Facter is a collection of several elements, and is written either as a simple value ("flat" fact) or as structured data ("structured" fact).

## Viewing fact values

If your Puppet masters are configured to use PuppetDB, you can view and search all of the facts for any node, including custom facts.

See the PuppetDB docs for more info.

# External facts

External facts provide a way to use arbitrary executables or scripts as facts, or set facts statically with structured data. With this information, you can write a custom fact in Perl, C, or a one-line text file.

### Fact locations

Distribute external facts with pluginsync. To add external facts to your Puppet modules, place them in `<MODULEPATH>/<MODULE>/facts.d/`.

If you're not using pluginsync, then external facts must go in a standard directory. The location of this directory varies depending on your operating system, whether your deployment uses Puppet Enterprise or open source releases, and whether you are running as root or Administrator. When calling Facter from the command line, you can specify the external facts directory with the `--external-dir` option.

**Note:** These directories don't necessarily exist by default; you might need to create them. If you create the directory, make sure to restrict access so that only administrators can write to the directory.

In a module (recommended):

```
<MODULEPATH>/<MODULE>/facts.d/
```

On Unix, Linux, or Mac OS X, there are three directories:

```
/opt/puppetlabs/facter/facts.d/
/etc/puppetlabs/facter/facts.d/
/etc/facter/facts.d/
```

On Windows:

```
C:\ProgramData\PuppetLabs\facter\facts.d\
```

When running as a non-root or non-Administrator user:

```
<HOME DIRECTORY>/.facter/facts.d/
```

**Note:** You can use custom facts as a non-root user only if you have first configured non-root user access and previously run Puppet agent as that same user.

### Executable facts on Unix

Executable facts on Unix work by dropping an executable file into the standard external fact path. A shebang (`#!`) is always required for executable facts on Unix. If the shebang is missing, the execution of the fact fails.

An example external fact written in Python:

```
#!/usr/bin/env python
data = {"key1" : "value1", "key2" : "value2" }

for k in data:
    print "%s=%s" % (k,data[k])
```

You must ensure that the script has its execute bit set:

```
chmod +x /etc/facter/facts.d/my_fact_script.py
```

For Facter to parse the output, the script must return key-value pairs on STDOUT in the format:

```
key1=value1
key2=value2
key3=value3
```

Using this format, a single script can return multiple facts.

### Executable facts on Windows

Executable facts on Windows work by dropping an executable file into the external fact path. Unlike with Unix, the external facts interface expects Windows scripts to end with a known extension. Line endings can be either `LF` or `CRLF`. The following extensions are supported:

- `.com` and `.exe`: binary executables
- `.bat` and `.cmd`: batch scripts
- `.ps1`: PowerShell scripts

As with Unix facts, each script must return key-value pairs on STDOUT in the format:

```
key1=value1
key2=value2
```

```
key3=value3
```

Using this format, a single script can return multiple facts in one return.

For batch scripts, the file encoding for the `.bat` or `.cmd` files must be ANSI or UTF8 without BOM, or you get strange output.

Here is a sample batch script which outputs facts using the required format:

```
@echo off
echo key1=val1
echo key2=val2
echo key3=val3
REM Invalid - echo 'key4=val4'
REM Invalid - echo "key5=val5"
```

For PowerShell scripts, the encoding used with `.ps1` files is flexible. PowerShell determines the encoding of the file at run time.

Here is a sample PowerShell script which outputs facts using the required format:

```
Write-Host "key1=val1"
Write-Host 'key2=val2'
Write-Host key3=val3
```

Save and execute this PowerShell script on the command line.

### Structured data facts

Facter can parse structured data files stored in the external facts directory and set facts based on their contents.

Structured data files must use one of the supported data types and must have the correct file extension. Facter supports the following extensions and data types:

- `.yaml`: YAML data, in the following format:

  ```
  ---
  key1: val1
  key2: val2
  key3: val3
  ```

- `.json`: JSON data, in the following format:

  ```
  {
      "key1": "val1",
      "key2": "val2",
      "key3": "val3"
  }
  ```

- `.txt`: Key-value pairs, of the `String` data type, in the following format:

  ```
  key1=value1
  key2=value2
  key3=value3
  ```

As with executable facts, structured data files can set multiple facts at one time.

```
{
  "datacenter":
  {
    "location": "bfs",
    "workload": "Web Development Pipeline",
    "contact": "Blackbird"
```

```
  },
  "provision":
  {
    "birth": "2017-01-01 14:23:34",
    "user": "alex"
  }
}
```

#### Structured data facts on Windows

All of the above types are supported on Windows with the following notes:

- The line endings can be either `LF` or `CRLF`.
- The file encoding must be either ANSI or UTF8 without BOM.

#### Troubleshooting

If your external fact is not appearing in Facter's output, running Facter in debug mode can reveal why and tell you which file is causing the problem:

```
# puppet facts --debug
```

One possible cause is a fact that returns invalid characters. For example if you used a hyphen instead of an equals sign in your script `test.sh`:

```
#!/bin/bash

echo "key1-value1"
```

Running `puppet facts --debug` yields the following message:

```
...
Debug: Facter: resolving facts from executable file "/tmp/test.sh".
Debug: Facter: executing command: /tmp/test.sh
Debug: Facter: key1-value1
Debug: Facter: ignoring line in output: key1-value1
Debug: Facter: process exited with status code 0.
Debug: Facter: completed resolving facts from executable file "/tmp/
test.sh".
...
```

If you find that an external fact does not match what you have configured in your `facts.d` directory, make sure you have not defined the same fact using the external facts capabilities found in the `stdlib` module.

#### Drawbacks

While external facts provide a mostly-equal way to create variables for Puppet, they have a few drawbacks:

- An external fact cannot internally reference another fact. However, due to parse order, you can reference an external fact from a Ruby fact.
- External executable facts are forked instead of executed within the same process.

#### Related information

Custom facts walkthrough on page 448

You can add custom facts by writing snippets of Ruby code on the Puppet master. Puppet then uses plug-ins in modules to distribute the facts to the client.

## Configuring Facter with facter.conf

The `facter.conf` file is a configuration file that allows you to cache and block fact groups, and manage how Facter interacts with your system. There are three sections: `facts`, `global`, and `cli`. All sections are optional and can be listed in any order within the file.

When you run Facter from the Ruby API, only the `facts` section and limited `global` settings are loaded.

Example `facter.conf` file:

```
facts : {
    blocklist : [ "file system", "EC2" ],
    ttls : [
        { "timezone" : 30 days },
    ]
}
global : {
    external-dir     : [ "path1", "path2" ],
    custom-dir       : [ "custom/path" ],
    no-exernal-facts : false,
    no-custom-facts  : false,
    no-ruby          : false
}

cli : {
    debug     : false,
    trace     : true,
    verbose   : false,
    log-level : "warn"
}
```

### Location

Facter does not create the `facter.conf` file automatically, so you must create it manually, or use a module to manage it. Facter loads the file by default from `/etc/puppetlabs/facter/facter.conf` on *nix systems and `C:\ProgramData\PuppetLabs\facter\etc\facter.conf` on Windows. Or you can specify a different default with the `--config` command line option:

```
facter --config path/to/my/config/file/facter.conf
```

### facts

The `facts` section of `facter.conf` contains settings that affect fact groups. A fact group is a set of individual facts that are resolved together because they all rely on the same underlying system information. When you add a group name to the config file as a part of either of these `facts` settings, all facts in that group will be affected. Currently only built-in facts can be cached or blocked.

The settings in this section are:

- `blocklist` — Prevents all facts within the listed groups from being resolved when Facter runs. Use the `--list-block-group` command line option to list valid groups.
- `ttls` — Caches the key-value pairs of groups and their duration to be cached. Use the `--list-cache-group` command line option to list valid groups.
    - Cached facts are stored as JSON in `/opt/puppetlabs/facter/cache/cached_facts` on *nix and `C:\ProgramData\PuppetLabs\facter\cache\cached_facts` on Windows.

Caching and blocking facts is useful when Facter is taking a long time and slowing down your code. When a system has a lot of something — for example, mount points or disks — Facter can take a long time to collect the facts from each one. When this is a problem, you can speed up Facter's collection by either blocking facts you're uninterested in (`blocklist`), or caching ones you don't need retrieved frequently (`ttls`).

To see a list of valid group names, from the command line, run `facter --list-block-groups` or `facter --list-cache-groups`. The output shows the fact group at the top level, with all facts in that group nested below:

```
$ facter --list-block-groups
EC2
  - ec2_metadata
  - ec2_userdata
file system
  - mountpoints
  - filesystems
  - partitions
```

If you want to block any of these groups, add the group name to the `facts` section of `facter.conf`, with the `blocklist` setting:

```
facts : {
    blocklist : [ "file system" ],
}
```

Here, the `file system` group has been added, so the `mountpoints`, `filesystems`, and `partitions` facts will all be prevented from loading.

### `global`

The `global` section of `facter.conf` contains settings to control how Facter interacts with its external elements on your system.

| Setting | Effect | Default |
| --- | --- | --- |
| `external-dir` | A list of directories to search for external facts. | |
| `custom-dir` | A list of directories to search for custom facts. | |
| `no-external*` | If `true`, prevents Facter from searching for external facts. | `false` |
| `no-custom*` | If `true`, prevents Facter from searching for custom facts. | `false` |
| `no-ruby*` | If `true`, prevents Facter from loading its Ruby functionality. | `false` |

*Not available when you run Facter from the Ruby API.

### `cli`

The `cli` section of `facter.conf` contains settings that affect Facter's command line output. All of these settings are ignored when you run Facter from the Ruby API.

| Setting | Effect | Default |
| --- | --- | --- |
| `debug` | If `true`, Facter outputs debug messages. | `false` |

| Setting | Effect | Default |
|---|---|---|
| `trace` | If `true`, Facter prints stacktraces from errors arising in your custom facts. | `false` |
| `verbose` | If `true`, Facter outputs its most detailed messages. | `false` |
| `log-level` | Sets the minimum level of message severity that gets logged. Valid options: `"none"`, `"fatal"`, `"error"`, `"warn"`, `"info"`, `"debug"`, `"trace"`. | `"warn"` |

# Resource types

Every resource (file, user, service, package, and so on) is associated with a resource type within the Puppet language. The resource type defines the kind of configuration it manages. This section provides information about the resource types that are built into Puppet.

- All resource types
- Core types cheat sheet on page 460

This page provides a reference guide for the core Puppet types: `package`, `file`, `service`, `notify`, `exec`, `cron`, `user`, and `group`.
- Optional resource types for Windows on page 465

In addition to the resource types included with Puppet, you can install custom resource types as modules from the Forge. This is especially useful when managing Windows systems, because there are several important Windows-specific resource types that are developed as modules rather than as part of core Puppet.
- exec
- Using exec on Windows on page 465

Puppet uses the same `exec` resource type on both *nix and Windows systems, and there are a few Windows-specific best practices and tips to keep in mind.
- file
- Using file on Windows on page 466

Use Puppet's built-in `file` resource type to manage files and directories on Windows, including ownership, group, permissions, and content, with the following Windows-specific notes and tips.
- filebucket
- group
- Using user and group on Windows on page 468

Use the built-in `user` and `group` resource types to manage user and group accounts on Windows.
- index
- notify
- package
- Using package on Windows on page 470

The built-in `package` resource type handles many different packaging systems on many operating systems, so not all features are relevant everywhere. This page offers guidance and tips for working with `package` on Windows.
- resources
- schedule
- service

- Using service on page 472

Puppet can manage services on nearly all operating systems.This page offers operating system-specific advice and best practices for working with `service`.

- stage
- tidy
- user

# Core types cheat sheet

This page provides a reference guide for the core Puppet types: `package`, `file`, `service`, `notify`, `exec`, `cron`, `user`, and `group`.

For detailed information about these types, see the Resource type reference or the other pages in this section.

## The trifecta: package, `file`, and `service`

Package, file, service: Learn it, live it, love it. Even if this is the only Puppet you know, you can get a lot done.

```
package { 'openssh-server':
  ensure => installed,
}

file { '/etc/ssh/sshd_config':
  source  => 'puppet:///modules/sshd/sshd_config',
  owner   => 'root',
  group   => 'root',
  mode    => '0640',
  notify  => Service['sshd'], # sshd restarts whenever you edit this file.
  require => Package['openssh-server'],
}

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

### package

Manages software packages.

| Attribute | Description | Notes |
| --- | --- | --- |
| `name` | The name of the package, as known to your packaging system. | Defaults to title. |

| Attribute | Description | Notes |
|---|---|---|
| `ensure` | Whether the package should be installed, and what version to use. | Allowed values:<br>• `present`<br>• `latest` (implies `present`)<br>• Any version string (implies `present`)<br>• `absent`<br>• `purged`<br>**CAUTION:** `purged` ensures `absent`, and deletes configuration files and dependencies, including those that other packages depend on. Provider-dependent. |
| `source` | Where to obtain the package, if your system's packaging tools don't use a repository. | |
| `provider` | Which packaging system to use (such as Yum or Rubygems), if a system has more than one available. | |

### file

Manages files, directories, and symlinks.

| Attribute | Description | Notes |
|---|---|---|
| `ensure` | Whether the file should exist, and what it should be. | Allowed values:<br>• `file`<br>• `directory`<br>• `link` (symlink)<br>• `present` (anything)<br>• `absent` |
| `path` | The full path to the file on disk. | Defaults to title. |
| `owner` | By name or UID. | |
| `group` | By name or GID. | |
| `mode` | Must be specified exactly. Does the right thing for directories. | |

For normal files:

| | |
|---|---|
| `source` | Where to download contents for the file. Usually a `puppet:///` URL. |
| `content` | The file's desired contents, as a string. Most useful when paired with templates, but you can also use the output of the `file` function. |

For directories:

| | |
|---|---|
| `source` | Where to download contents for the directory, when `recurse => true`. |
| `recurse` | Whether to recursively manage files in the directory. |
| `purge` | Whether unmanaged files in the directory should be deleted, when `recurse => true`. |

For symlinks:

| | |
|---|---|
| `target` | The symlink target. (Required when `ensure => link`.) |

Other notable attributes:

- `backup`
- `checksum`
- `force`
- `ignore`
- `links`
- `recurselimit`
- `replace`

### service

Manages services running on the node. As with packages, some platforms have better tools than others, so read the relevant documentation before you begin.

You can make services restart whenever a file changes with the `subscribe` or `notify` metaparameters. For more info, see Relationships and ordering.

| Attribute | Description | Notes |
|---|---|---|
| `name` | The name of the service to run. | Defaults to title. |
| `ensure` | The desired status of the service. | Allowed values:<br>• `running` (or `true`)<br>• `stopped` (or `false`) |
| `enable` | Whether the service should start on boot. Doesn't work on all systems. | |
| `hasrestart` | Whether to use the init script's restart command instead of stop+start. | Defaults to false. |
| `hasstatus` | Whether to use the init script's status command. | Defaults to true. |

Other notable attributes:

If a service has a bad init script, you can work around it and manage almost anything using the `status`, `start`, `stop`, `restart`, `pattern`, and `binary` attributes.

## Other core types

Beyond `package`, `file`, and `service`, these core types are among the most useful and commonly used.

### notify

Logs an arbitrary message, at the `notice` log level. This appears in the POSIX syslog or Windows Event Log on the agent node and is also logged in reports.

```
notify { "This message is getting logged on the agent node.": }
```

| Attribute | Description | Notes |
| --- | --- | --- |
| `message` | The message to log. | Defaults to title. |

### exec

Executes an arbitrary command on the agent node. When using execs, you must either make sure the command can be safely run multiple times, or specify that it should only run under certain conditions.

| Important attributes | Description | Notes |
| --- | --- | --- |
| `command` | The command to run. If this isn't a fully-qualified path, use the `path` attribute. | Defaults to title. |
| `path` | Where to look for executables, as a colon-separated list or an array. | |
| `returns` | Which exit codes indicate success. | Defaults to 0. |
| `environment` | An array of environment variables to set (for example, `['MYVAR=somevalue', 'OTHERVAR=othervalue']`). | |
| The following attributes limit when a command should run. | | |
| `creates` | A file to look for before running the command. The command only runs if the file doesn't exist. | |
| `refreshonly` | If `true`, the command runs only if a resource it subscribes to (or a resource which notifies it) has changed. | |
| `onlyif` | A command or array of commands; if any have a non-zero return value, the command won't run. | |
| `unless` | The opposite of `onlyif`. | |

Other notable attributes: `cwd`, `group`, `logoutput`, `timeout`, `tries`, `try_sleep`, `user`

### cron

Manages cron jobs. On Windows, use `scheduled_task` instead.

```
cron { 'logrotate':
  command => "/usr/sbin/logrotate",
  user    => "root",
  hour    => 2,
  minute  => 0,
}
```

| Important attributes | Description | Notes |
|---|---|---|
| `command` | The command to execute. | |
| `ensure` | Whether the job should exist. | Allowed values:<br>• `present`<br>• `absent` |
| `hour`, `minute`, `month`, `monthday`, `weekday` | The timing of the cron job. | |

Other notable attributes: `environment`, `name`, `special`, `target`, `user`

### user

Manages user accounts; mostly used for system users.

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

| Important Attributes | Description | Notes |
|---|---|---|
| `name` | The name of the user. | Defaults to title. |
| `ensure` | Whether the user should exist. | Allowed values:<br>• `present`<br>• `absent`<br>• `role` |
| `uid` | The user ID. Must be specified numerically; chosen automatically if omitted. | Read-only on Windows. |
| `gid` | The user's primary group. Can be specified numerically or by name. | Not used on Windows; use `groups` instead. |
| `groups` | An array of other groups to which the user belongs. | Don't include the group specified as the `gid`. |
| `home` | The user's home directory. | |
| `managehome` | Whether to manage the home directory when managing the user. | If you don't set this to `true`, you'll need to create the user's home directory manually. |
| `shell` | The user's login shell. | |

Other notable attributes: `comment`, `expiry`, `membership`, `password`, `password_max_age`, `password_min_age`, `purge_ssh_keys`, `salt`

### group

Manages groups.

| Important attributes | Description | Notes |
|---|---|---|
| `name` | The name of the group. | Defaults to title. |
| `ensure` | Whether the group should exist. | Allowed values:<br>• `present`<br>• `absent` |
| `gid` | The group ID; must be specified numerically, and is chosen automatically if omitted. | Read-only on Windows. |
| `members` | Users and groups that should be members of the group. | Only applicable to certain operating systems; see the full type reference for details. |

## Optional resource types for Windows

In addition to the resource types included with Puppet, you can install custom resource types as modules from the Forge. This is especially useful when managing Windows systems, because there are several important Windows-specific resource types that are developed as modules rather than as part of core Puppet.

If you're doing heavy management of Windows systems, the following modules (which are collected in the puppetlabs/windows module pack) might be helpful:

- puppetlabs/acl: A resource type for managing access control lists (ACLs) on Windows.
- puppetlabs/registry: A resource type for managing arbitrary registry keys.
- puppetlabs/reboot: A resource type for managing conditional reboots, which can be necessary for installing certain software.
- puppetlabs/dism: A resource type for enabling and disabling Windows features (on Windows 7 or 2008 R2 and newer).
- puppetlabs/powershell: An alternative `exec` provider that can directly execute PowerShell commands.

Other resource types created by community members are also available on the Forge. The best way to find new resource types is by searching for "Windows" on the Forge and exploring the results.

**Remember:** Plugins from the Forge might not have the same amount of quality assurance and test coverage as the core resource types included in Puppet.

## Using exec on Windows

Puppet uses the same `exec` resource type on both *nix and Windows systems, and there are a few Windows-specific best practices and tips to keep in mind.

Puppet can run binary files (such as `exe`, `com`, or `bat`), and can log the child process output and exit status. To ensure the resource is idempotent, specify one of the `creates`, `onlyif`, or `unless` attributes.

### Command extensions

If a file extension for the command is not specified (for example, `ruby` instead of `ruby.exe`), Puppet will use the `PATHEXT` environment variable to resolve the appropriate binary. `PATHEXT` is a Windows-specific variable that lists the valid file extensions for executables.

### Exit codes

On Windows, most exit codes should be integers between 0 and 2147483647.

Larger exit codes on Windows behave inconsistently across different tools. The Win32 APIs define exit codes as 32-bit unsigned integers, but both the cmd.exe shell and the .NET runtime cast them to signed integers. This means some tools will report negative numbers for exit codes above 2147483647. For example, cmd.exe reports 4294967295 as -1.

Because Puppet uses the GetExitCodeProcess Win32 API, it reports the very large number instead of the negative number, which might not be what you expect if you got the exit code from a cmd.exe session.

Microsoft recommends against using negative or very large exit codes, and you should avoid them when possible.

**Tip:** To convert a negative exit code to the positive one Puppet will use, subtract it from 4294967296.

### Shell built-ins

Puppet does not support a shell provider for Windows, so if you want to execute shell built-ins (such as `echo`), you must provide a complete `cmd.exe` invocation as the command. For example, `command => 'cmd.exe /c echo "hello"'`.

When using `cmd.exe` and specifying a file path in the command line, be sure to use backslashes. For example, `'cmd.exe /c type c:\path\to\file.txt'`. If you use forward slashes, `cmd.exe` returns an error.

### Optional PowerShell exec provider

An optional PowerShell `exec` provider is available as a plug-in and is helpful if you need to run PowerShell commands from within Puppet. To use it, install puppetlabs/powershell.

### Inline PowerShell scripts

If you choose to execute PowerShell scripts using the default Puppet `exec` provider on Windows, you must specify the `remotesigned` execution policy as part of the `powershell.exe` invocation:

```
exec { 'test':
  command => 'C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -
executionpolicy remotesigned -file C:\test.ps1',
}
```

## Using `file` on Windows

Use Puppet's built-in `file` resource type to manage files and directories on Windows, including ownership, group, permissions, and content, with the following Windows-specific notes and tips.

```
file { 'c:/mysql/my.ini':
  ensure => 'file',
  mode   => '0660',
  owner  => 'mysql',
  group  => 'Administrators',
  source => 'N:/software/mysql/my.ini',
}
```

### Take care with backslashes in file paths

The issue of backslashes and forward-slashes in file paths can get complicated. See Handling file paths on Windows for more information.

### Be consistent with capitalization in file names

If you refer to a file resource in multiple places in a manifest (such as when creating relationships between resources), be consistent with the capitalization of the file name. If you use `my.ini` in one place, don’t use `MY.INI` in another place.

Windows NTFS filesystems are case-insensitive (albeit case-preserving); Puppet is case-sensitive. Windows itself won't be confused by inconsistent case, but Puppet will think you're referring to different files.

### Make sure the Puppet user account has appropriate permissions

To manage files properly, Puppet needs the following Windows privileges:

- Create symbolic links
- Back up files and directories
- Restore files and directories

When Puppet runs as a service, make sure its user account is a member of the local `Administrators` group. When you use the `PUPPET_AGENT_ACCOUNT_USER` parameter with the MSI installer, the user will automatically be added to the `Administrators` group.

Before running Puppet interactively (on Windows Vista or 2008 and later versions), start the command prompt window with elevated privileges by right-clicking on the start menu and choosing "Run as Administrator."

### Managing file permissions: the `mode` attribute and the acl module

The permissions models used by *nix and Windows are quite different. When you use the `mode` attribute, the `file` type manages them both like *nix permissions, and translates the mode to roughly equivalent access controls on Windows. This makes basic controls fairly simple, but doesn't work for managing complex access rules.

If you need fine-grained Windows access controls, use the `puppetlabs/acl` module, which provides an optional `acl` resource type that manages permissions in a Windows-centric way. Leave `mode` unspecified and add an `acl` resource. See the `acl` module's documentation for details.

### How *nix modes map to Windows permissions

*nix permissions are expressed as either a quoted octal number (such as `"755"`), or a string of symbolic modes, (such as `"u=rwx,g=rx,o=rx"`). See the reference for the `file` type's `mode` attribute for more details about the syntax.

These mode expressions generally manage three kinds of permission — read, write, execute — for three kinds of user — owner, group, other. They translate to Windows permissions as follows:

- The read, write, and execute permissions are interpreted as the `FILE_GENERIC_READ`, `FILE_GENERIC_WRITE`, and `FILE_GENERIC_EXECUTE` access rights, respectively.
- The `Everyone` SID is used to represent users other than the owner and group.
- Directories on Windows can have the sticky bit, which makes it so users can delete files only if they own the containing directory.
- The owner of a file can be a group (for example, `owner => 'Administrators'`) and the group of a file can be a user (for example, `group => 'Administrator'`).
- While it's possible for the owner and group to be the same, this is strongly discouraged. Doing so can cause problems when the mode gives different permissions to the owner and group (such as `0750`).
- The group can't have higher permissions than the owner. Other users can't have higher permissions than the owner or group. In other words, `0640` and `0755` are supported, but `0460` is not.

### Extra behavior when managing permissions with `mode`

When you manage permissions with the `mode` attribute, it has the following side effects:

- The owner of a file or directory always has the `FULL_CONTROL` access right.
- The security descriptor is always set to protected. This prevents the file from inheriting more permissive access controls from the directory that contains it.

### File sources

The `source` attribute of a file can be a Puppet URL, a local path, a UNC path, or a path to a file on a mapped drive.

### Handling line endings

Windows usually uses CRLF line endings, rather than the LF line endings used by *nix. In most cases, Puppet **does not** automatically convert line endings when managing files on Windows.

If a file resource uses the `content` or `source` attributes, Puppet writes the file in binary mode, using the line endings that are present in the content. If the manifest, template, or source file is saved with CRLF line endings, Puppet uses those endings in the destination file.

Non-`file` resource types that make partial edits to a system file (most notably the `host` resource type, which manages the `%windir%\system32\drivers\etc\hosts` file) manage their files in text mode, and automatically translate between Windows and *nix line endings.

**Note:** When writing your own resource types, you can get this same behavior by using the `flat` file type.

## Using `user` and `group` on Windows

Use the built-in `user` and `group` resource types to manage user and group accounts on Windows.

### Managing local user and group resources

Puppet uses the `user` and `group` resource types to manage local accounts. You can't write a Puppet resource that describes a domain user or group. However, a local `group` resource can manage which domain accounts belong to the local group.

### Managing group membership with Puppet

Windows manages group membership by specifying the groups to which a user belongs, or by specifying the members of a group. Puppet supports both of these methods.

When Puppet is managing a local user, you can list the groups that the user belongs to. These groups can be a local group account (such as `Administrators`) or a domain group account.

When Puppet is managing a local group, you can list the members that belong to the group. Each member can be a local account (such as `Administrator`) or a domain account, where each account can be a user or a group account.

When managing a user, Puppet makes sure that the user belongs to all of the groups listed in the manifest. If the user belongs to a group not specified in the manifest, Puppet does not remove the user from the group.

If you want to ensure that a user belongs to only the groups listed in the manifest, and no others, specify the `membership` attribute for the user. If set to `inclusive`, Puppet removes the user from any group not listed in the manifest.

Similarly, when managing a group, Puppet makes sure all of the members listed in the manifest are added to the group. Existing members of the group who are not listed in the manifest are ignored.

To ensure that a group contains only the members listed in the manifest, and no others, specify the `auth_membership` attribute for the group. When this attribute is present and set to `true`, Puppet removes any members of the group not listed in the manifest.

### Allowed `user` attributes on Windows

When managing Windows user accounts, you can use the following `user` resource type attributes:

| Attribute | Usage notes |
| --- | --- |
| `name` | |

| Attribute | Usage notes |
| --- | --- |
| `ensure` | |
| `comment` | |
| `groups` | You cannot use the `gid` attribute with Windows. |
| `home` | |
| `managehome` | |
| `membership` | |
| `password` | Passwords must be specified in cleartext, because Windows does not have an API for setting the password hash. |
| `auth_membership` | |
| `uid` | **Read-only.** Available for inspecting a user by running `puppet resource user <NAME>`. The `uid` value will be the user's SID (see below). |

### Allowed `group` attributes on Windows

When managing Windows group accounts, you can use the following `group` resource type attributes:

| Attribute | Usage notes |
| --- | --- |
| `name` | |
| `ensure` | |
| `members` | |
| `auth_membership` | |
| `gid` | **Read-only.** Available for inspecting a group by running `puppet resource group <NAME>`. The `gid` value will be the group's SID (see below). |

### Names and security identifiers (SIDs)

On Windows, user and group account names can take multiple forms, such as:

- `Administrators`
- `<host>\Administrators`
- `BUILTIN\Administrators`
- `S-1-5-32-544`

The `S-1-5-32-544` name form is called a security identifier (SID). Puppet treats all these forms equally: when comparing two account names, it transforms account names into their canonical SID form and compares the SIDs.

When you refer to a user or group in multiple places in a manifest (such as when creating relationships between resources), be consistent with how you capitalize the name. Names are case-sensitive in Puppet manifests, but case-insensitive on Windows. It's important that the cases match, however, because autorequire will attempt to match users with fully qualified names (such as `User[BUILTIN\Administrators]`) in addition to SIDs (such as `User[S-1-5-32-544]`). It might not match in cases where domain accounts and local accounts have the same name, such as `Domain\Bob` versus `LOCAL\Bob`.

**Note:** When listed for reporting or by `puppet resource`, groups always return the fully qualified form when describing a user, such as `BUILTIN\Administrators`. These fully qualified names might not look the same as in the names specified in the manifest.

## Using package on Windows

The built-in `package` resource type handles many different packaging systems on many operating systems, so not all features are relevant everywhere. This page offers guidance and tips for working with `package` on Windows.

```
package { 'mysql':
  ensure          => '5.5.16',
  source          => 'N:\packages\mysql-5.5.16-winx64.msi',
  install_options => ['INSTALLDIR=C:\mysql-5.5'],
}

package { "Git version 1.8.4-preview20130916":
 ensure          => installed,
 source          => 'C:\code\puppetlabs\temp\windowsexample\Git-1.8.4-
preview20130916.exe',
 install_options => ['/VERYSILENT']
}
```

### Supported package types: MSI and EXE

Puppet can install and remove MSI packages and executable installers on Windows. Both package types use the default `windows` package provider.

Alternatively, a Chocolatey package provider is available on the Forge.

### The `source` attribute is required

When managing packages using the `windows` package provider, you must specify a package file using the `source` attribute.

The source can be a local file or a file on a mapped network drive. For MSI installers, you can use a UNC path. Puppet URLs are not supported for the `package` type's `source` attribute, but you can use `file` resources to copy packages to the local system. The `source` attribute accepts both forward- and backslashes.

### The package name must be the `DisplayName`

The title (or `name`) of the package must match the value of the package's `DisplayName` property in the registry, which is also the value displayed in the **Add/Remove Programs** or **Programs and Features** control panels.

If the provided name and the installed name don't match, Puppet assumes the package is not installed and tries to install it again.

To determine a package's `DisplayName`:

1. Install the package on an example system.
2. Run `puppet resource package` to see a list of installed packages.
3. Copy the name of the package from the list.

Some packages (Git is a notable example) change their display names with every newly released version. See the section below on handling package versions and upgrades.

### Install and uninstall options

The Windows package provider also supports package-specific `install_options` (such as install directory) and `uninstall_options`. These options vary across packages, so see the documentation for the specific package you're installing. Options are specified as an array of strings or hashes.

MSI properties can be specified as an array of strings following the `property=key` pattern; use one string per property. Command line flags to executable installers can be specified as an array of strings, with one string per flag.

Any file path arguments within the `install_options` attribute (such as `INSTALLDIR`) should use backslashes (\), not forward slashes. Be sure to escape your backslashes appropriately. For more info, see Handling file paths on Windows.

Use the hash notation for file path arguments because they might contain spaces. For example:

```
install_options => [ { 'INSTALLDIR' => ${packagedir} } ]
```

## Handling versions and upgrades

Setting `ensure => latest` (which requires the `upgradeable` feature) doesn't work on Windows, because Windows doesn't have central package repositories like on most *nix systems.

There are two ways to handle package versions and upgrades on Windows.

### Packages with real versions

Many packages on Windows have version metadata. To tell whether a package has usable version metadata, install it on a test system and use `puppet resource package` to inspect it.

To upgrade these packages, replace the `source` file and set `ensure => '<VERSION>'`. For example:

```
package { 'mysql':
  ensure          => '5.5.16',
  source          => 'N:\packages\mysql-5.5.16-winx64.msi',
  install_options => ['INSTALLDIR=C:\mysql-5.5'],
}
```

The next time Puppet runs, it will notice that the versions don't match and will install the new package. This makes the installed version match the new version, so Puppet won't attempt to re-install the package until you change the version in the manifest again.

The version you use in `ensure` must exactly match the version string that the package reports when you inspect it with `puppet resource`. If it doesn't match, Puppet will repeatedly try to install it.

### Packages that include version info in their `DisplayName`

Some packages don't embed version metadata; instead, they change their `DisplayName` property with each release. The Git packages are a notable example.

To upgrade these packages, replace the `source` file and update the resource's title or `name` to the new `DisplayName`. For example:

```
package { "Git version 1.8.4-preview20130916":
 ensure          => installed,
 source          => 'C:\code\puppetlabs\temp\windowsexample\Git-1.8.4-
preview20130916.exe',
 install_options => ['/VERYSILENT']
}
```

The next time Puppet runs, it will notice that the names don't match and will install the new package. This makes the installed name match the new name, so Puppet won't attempt to re-install the package until you change the name in the manifest again.

The name you use in the title must exactly match the name that the package reports when you inspect it with `puppet resource`. If it doesn't match, Puppet will repeatedly try to install it.

# Using `service`

Puppet can manage services on nearly all operating systems.This page offers operating system-specific advice and best practices for working with `service`.

## Using `service` on *nix systems

If your *nix operating system has a good system for managing services, and all the services you care about have working init scripts or service configs, you can write small `service` resources with just the `ensure` and `enable` attributes.

For example:

```
service { 'apache2':
  ensure => running,
  enable => true,
}
```

**Note:** Some *nix operating systems don't support the enable attribute.

### Defective init script

On platforms that use SysV-style init scripts, Puppet assumes the script will have working `start`, `stop`, and `status` commands.

If the `status` command is missing, set `hasstatus => false` for that service. This makes Puppet search the process table for the service's name to check whether it's running.

In some rare cases — such as virtual services like the Red Hat `network` — a service won't have a matching entry in the process table. If a service acts like this and is also missing a status command, set `hasstatus => false` and also specify either `status` or `pattern` attribute.

### No init script or service config

If some of your services lack init scripts, Puppet can compensate, as in the following example:

```
service { "apache2":
  ensure  => running,
  start   => "/usr/sbin/apachectl start",
  stop    => "/usr/sbin/apachectl stop",
  pattern => "/usr/sbin/httpd",
}
```

In addition to specifing `ensure`, specify also how to start the service, how to stop it, how to check whether it's running, and optionally how to restart it.

| | |
|---|---|
| **Start** | Use either `start` or `binary` to specify a start command. The difference is that `binary` also gives you default behavior for `stop` and `status`. |
| **Stop** | If you specified `binary`, Puppet defaults to finding that same executable in the process table and killing it.<br>If the service should be stopped some other way, use the `stop` attribute to specify a command. |
| **Status** | If you specified `binary`, Puppet checks for that executable in the process table. If it doesn't find it, it reports that the service isn't running. |

| | |
|---|---|
| | If there's a better way to check the service's status, or if the `start` command is just a script and a different process implements the service itself, use either `status` (a command that exits 0 if the service is running and nonzero otherwise) or `pattern` (a pattern to search the process table for). |
| **Restart** | If a service needs to be reloaded, Puppet defaults to stopping it and starting it again. If you have a safer command for restarting a service, you can optionally specify it in the `restart` attribute. |

### Using `service` on macOS

macOS handles services much like most *nix-based systems. The main difference is that `enable` and `ensure` are much more closely linked — running services are always enabled, and stopped ones are always disabled.

For best results, either leave `enable` blank or make sure it's set to `true` whenever `ensure => running`.

The launchd plists that configure your services must be in one of the following directories:

- `/System/Library/LaunchDaemons`
- `/System/Library/LaunchAgents`
- `/Library/LaunchDaemons`
- `/Library/LaunchAgents`

You can also specify `start` and `stop` commands to assemble your own services, much like on *nix systems.

### Using `service` on Windows

On Windows, Puppet can start, stop, enable, disable, list, query, and configure services. It expects that all services will run with the built-in Service Control Manager (SCM) system. It does not support configuring service dependencies, the account to run as, or desktop interaction.

When writing `service` resources for Windows, remember the following:

- Use the short service name (such as `wuauserv`) in Puppet, not the display name (such as `Automatic Updates`).
- Set `enable => true` to assign a service the Automatic startup type.
- Set `enable => manual` to assign the Manual startup type.

For example, here is a complete service resource:

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

# Tracking Puppet activity with reports

Puppet creates a report about its actions and your infrastructure each time it applies a catalog during a Puppet run. You can create and use report processors to generate insightful information or alerts from those reports.

- Reporting on page 474

In a client-server configuration, an agent sends its report to the master for processing. In a standalone configuration, the `puppet apply` command processes the node's own reports. In both configurations, report processor plugins handle received reports. If you enable multiple report processors, Puppet runs all of them for each report.

- Built-in report processors

• Writing custom report processors on page 475
Create and use report processors to generate insightful information or alerts from Puppet reports. You can write your own report processor in Ruby and include it in a Puppet module. Puppet uses the processor to send report data to the service in the format you defined.
• Report format on page 476
Puppet versions 5.5.3 and later generate report format 10. This format is backward compatible with report format 9, which is used in Puppet versions 5.5.0 to 5.5.2.

## Reporting

In a client-server configuration, an agent sends its report to the master for processing. In a standalone configuration, the `puppet apply` command processes the node's own reports. In both configurations, report processor plugins handle received reports. If you enable multiple report processors, Puppet runs all of them for each report.

Each processor typically does one of two things with the report:

- It sends some of the report data to another service, such as PuppetDB, that can collate it.
- It triggers alerts on another service if the data matches a specified condition, such as a failed run.

That external service can then provide a way to view the processed report.

### Report data

Puppet report processors handle these points of data:

- Metadata about the node, its environment and Puppet version, and the catalog used in the run.
- The status of every resource.
- Actions, also called events, taken during the run.
- Log messages generated during the run.
- Metrics about the run, such as its duration and how many resources were in a given state.

That external service can then provide a way to view the processed report.

### Configuring reporting

An agent sends reports to the master by default. You can turn off reporting by changing the `report` setting in an agent's `puppet.conf` file.

On masters and on nodes running Puppet apply, you can configure enabled report processors as a comma-separated list in the `reports` setting. The default `reports` value is `'store'`, which stores reports in the configured `reportdir`.

To turn off reports entirely, set `reports` to `'none'`.

For details about configuration settings in Puppet, see the Configuration reference.

### Accessing reports

There are multiple ways to access Puppet report data:

- In Puppet Enterprise (PE), view run logs and event reports on the **Reports** page. See Infrastructure reports.
- In PuppetDB, with its report processor enabled, interface with third-party tools such as Puppetboard or PuppetExplorer.
- Use one of the built-in report processors. For example, the `http` processor sends YAML dumps of reports through POST requests to a designated URL; the log processor saves received logs to a local `log` file.
- Use a report processor from a module, such as `tagmail`.
- Query PuppetDB for stored report data and build your own tools to display it. For details about the types of data that PuppetDB collects and the API endpoints it uses, see the API documentation for the endpoints `events`, `event-counts`, and `aggregate-event-counts`.

- Write a custom report processor.

#### Related information

puppet.conf: The main config file on page 65
The `puppet.conf` file is Puppet's main config file. It configures all of the Puppet commands and services, including Puppet agent, Puppet master, Puppet apply, and `puppetserver ca`. Nearly all of the settings listed in the configuration reference can be set in `puppet.conf`.

Writing custom report processors on page 475
Create and use report processors to generate insightful information or alerts from Puppet reports. You can write your own report processor in Ruby and include it in a Puppet module. Puppet uses the processor to send report data to the service in the format you defined.

Report format on page 476
Puppet versions 5.5.3 and later generate report format 10. This format is backward compatible with report format 9, which is used in Puppet versions 5.5.0 to 5.5.2.

## Writing custom report processors

Create and use report processors to generate insightful information or alerts from Puppet reports. You can write your own report processor in Ruby and include it in a Puppet module. Puppet uses the processor to send report data to the service in the format you defined.

A report processor must follow these rules:

- The processor name must be a valid Ruby symbol that starts with a letter and contains only alphanumeric characters.
- The processor must be in its own Ruby file, `<PROCESSOR_NAME>.rb`, and stored inside the Puppet module directory `lib/puppet/reports/`
- The processor code must start with `require 'puppet'`
- The processor code must call the method `Puppet::Reports.register_report(:NAME)` This method takes the name of the report as a symbol, and a mandatory block of code with no arguments that contains:
  - A Markdown-formatted string describing the processor, passed to the `desc(<DESCRIPTION>)` method.
  - An implementation of a method named `process` that contains the report processor's main functionality.

Puppet lets the `process` method access a `self` object, which will be a `Puppet::Transaction::Report` object describing a Puppet run.

The processor can access report data by calling accessor methods on `self`, and it can forward that data to any service you configure in the report processor. It can also call `self.to_yaml` to dump the entire report to YAML. Note that the YAML output isn't a safe, well-defined data format — it's a serialized object.

#### Example report processor

To use this report processor, include it in the comma-separated list of processors in the Puppet master's `reports` setting in `puppet.conf`: `reports = store,myreport`.

```
# Located in /etc/puppetlabs/puppet/modules/myreport/lib/puppet/reports/
myreport.rb.
require 'puppet'
# If necessary, require any other Ruby libraries for this report here.

Puppet::Reports.register_report(:myreport) do
  desc "Process reports via the fictional my_cool_cmdb API."

  # Declare and configure any settings here. We'll pretend this connects to
 our API.
  my_api = MY_COOL_CMD
```

```
  # Define and configure the report processor.
  def process
    # Do something that sets up the API we're sending the report to here.
    # For instance, let's check on the node's status using the report object
 (self):
    if self.status != nil then
      status = self.status
    else
      status = 'undefined'
    end

    # Next, let's do something if the status equals 'failed'.
    if status == 'failed' then
      # Finally, dump the report object to YAML and post it using the API
 object:
      my_api.post(self.to_yaml)
    end
  end
end
```

To use this report processor, include it in the comma-separated list of processors in the Puppet master's `reports` setting in `puppet.conf`:

```
reports = store,myreport
```

For more examples using this API, see the built-in reports' source code or one of these custom reports created by a member of the Puppet community:

- Report failed runs to Jabber/XMPP
- Send metrics to a Ganglia server via gmetric

These community reports aren't provided or supported by Puppet, Inc.

#### Related information

Report format on page 476
Puppet versions 5.5.3 and later generate report format 10. This format is backward compatible with report format 9, which is used in Puppet versions 5.5.0 to 5.5.2.

## Report format

Puppet versions 5.5.3 and later generate report format 10. This format is backward compatible with report format 9, which is used in Puppet versions 5.5.0 to 5.5.2.

### `Puppet::Transaction::Report`

| Property | Type | Description |
|---|---|---|
| `host` | string | The host that generated this report. |
| `time` | datetime | When the Puppet run began. |
| `logs` | array | Zero 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` |

| Property | Type | Description |
| --- | --- | --- |
| `configuration_version` | string or integer | The configuration version of the Puppet run. This is a string for user-specified versioning schemes. Otherwise it is an integer representing seconds since the Unix epoch. |
| `transaction_uuid` | string | A UUID covering the transaction. The query parameters for the catalog retrieval include the same UUID. |
| `code_id` | string | The ID of the code input to the compiler. |
| `job_id` | string, or null | The ID of the job in which this transaction occurred. |
| `catalog_uuid` | string | A master generated catalog UUID, useful for connecting a single catalog to multiple reports. |
| `master_used` | string | The name of the master used to compile the catalog. If failover occurred, this holds the first master successfully contacted. If this run had no master (for example, a `puppet apply` run), this field is blank. |
| `report_format` | string or integer | `"10"` or `10` |
| `puppet_version` | string | The version of the Puppet agent. |
| `status` | string | The transaction status: `failed`, `changed`, or `unchanged`. |
| `transaction_completed` | Boolean | Whether the transaction completed. For instance, if the transaction had an unrescued exception, `transaction_completed = false`. |
| `noop` | Boolean | Whether the Puppet run was in no-operation mode when it ran. |
| `noop_pending` | Boolean | Whether there are changes that were not applied because of no-operation mode. |
| `environment` | string | The environment that was used for the Puppet run. |
| `corrective_change` | Boolean | True if a change or no-operation event in this report was caused by an unexpected change to the system between Puppet runs. |
| `cached_catalog_status` | string | The status of the cached catalog used in the run: `not_used`, `explicitly_requested`, or `on_failure`. |

### `Puppet::Util::Log`

| Property | Type | Description |
|---|---|---|
| `file` | string | The path and filename of the manifest file that triggered the log message. This property is not always present. |
| `line` | integer | The manifest file's line number that triggered the log message. This property is not always present. |
| `level` | symbol | The severity level of the message `:debug`, `:info`, `:notice`, `:warning`, `:err`, `:alert`, `:emerg`, `:crit`. |
| `message` | string | The text of the message. |
| `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 | The time at which the message was sent. |

### `Puppet::Util::Metric`

A `Puppet::Util::Metric` object represents all the metrics in a single category.

| Property | Type | Description |
|---|---|---|
| `name` | string | Specifies the 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 | The name of the metric formatted as a title. Underscores are replaced with spaces and the first word is capitalized. |
| `values` | array | All the metric values within this category. Each value is in the form `[name, label, value]`, where `name` is the particular metric as a string, `label` is the metric name formatted as a title, and `value` is the metric quantity as an integer or a float. |

The metrics that appear in a report are part of a fixed set and arranged in the following categories:

| | |
|---|---|
| **`time`** | Includes a metric for every resource type for which there is at least one resource in the catalog, plus two additional metrics: `config_retrieval` and `total`. Each value in the `time` category is a float. |

| | |
|---|---|
| | In an inspect report, there is an additional `inspect` metric. |
| **`resources`** | Includes the metrics `failed`, `out_of_sync`, `changed`, and `total`. Each value in the `resources` category is an integer. |
| **`events`** | Includes up to five metrics: `success`, `failure`, `audit`, `noop`, and `total`. `total` is always present; the others are present when their values are non-zero. Each value in the `events` category is an integer. |
| **`changes`** | Includes one metric, `total`. Its value is an integer. |

**Note:** Failed reports contain no metrics.

### `Puppet::Resource::Status`

A `Puppet::Resource::Status` object represents the status of a single resource.

| Property | 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 that corresponds to this `Puppet::Resource::Status` object in the `resource_statuses` hash. Deprecated. |
| `provider_used` | string | The name of the provider used by the resource. |
| `file` | string | The path and filename of the manifest file that declared the resource. |
| `line` | integer | The line number in the manifest file that 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 that changed. Always `0` in inspect reports. |
| `out_of_sync_count` | integer | The number of properties that 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. |

| Property | Type | Description |
| --- | --- | --- |
| `events` | array | The `Puppet::Transaction::Event` objects for the resource. |
| `out_of_sync` | Boolean | True when `out_of_sync_count > 0`, otherwise false. Deprecated. |
| `changed` | Boolean | True when `change_count > 0`, otherwise false. Deprecated. |
| `skipped` | Boolean | True when the resource was skipped, otherwise false. |
| `failed` | Boolean | True when Puppet experienced an error while evaluating this resource, otherwise false. Deprecated. |
| `failed_to_restart` | Boolean | True when Puppet experienced an error while trying to restart this resource, for example, when a Service resource has been notified from another resource. |
| `containment_path` | array | An array of strings; each element represents a container (type or class) that, together, make up the path of the resource in the catalog. |

### `Puppet::Transaction::Event`

A `Puppet::Transaction::Event` object represents a single event for a single resource.

| Property | Type | Description |
| --- | --- | --- |
| `audited` | Boolean | True when this property is being audited, otherwise false. True in inspect reports. |
| `property` | string | The property for which the event occurred. This value is missing if the provider errored out before it could be determined. |
| `previous_value` | string, array, or hash | The value of the property before the change (if any) was applied. This value is missing if the provider errored out before it could be determined. |
| `desired_value` | string, array, or hash | The value specified in the manifest. Absent in inspect reports. This value is missing if the provider errored out before it could be determined. |
| `historical_value` | string, array, or hash | The audited value from a previous run of Puppet, if known. Otherwise nil. Absent in inspect reports. This value is missing if the provider errored out before it could be determined. |

| Property | Type | Description |
|---|---|---|
| `message` | string | The log message generated by this event. |
| `name` | symbol | The name of the event. Absent in inspect reports. |
| `status` | string | The event status:<br>• `success`: Property was out of sync and was successfully changed to be in sync.<br>• `failure`: Property was out of sync and couldn't be changed to be in sync due to an error.<br>• `noop`: Property was out of sync but wasn't changed because the run was in no-operation mode.<br>• `audit`: Property was in sync and was being audited. Inspect reports are always in `audit` status. |
| `redacted` | Boolean | Whether this event has been redacted. |
| `time` | datetime | The time at which the property was evaluated. |
| `corrective_change` | Boolean | True if this event was caused by an unexpected change to the system between Puppet runs. |

### Differences from report format 9

- `failed_to_restart` was added to `Puppet::Resource::Status`

# Writing external node classifiers

An external node classifier (ENC) is a script or application that tells Puppet which classes a node should have. It can replace or work in concert with the node definitions in the main site manifest (`site.pp`).

Depending on the external data sources you use in your infrastructure, building an external node classifier can be a valuable way to extend Puppet.

### External node classifiers

An external node classifier is an executable that Puppet Server or `puppet apply` can call; it doesn't have to be written in Ruby. Its only argument is the name of the node to be classified, and it returns a YAML document describing the node.

Inside the ENC, you can reference any data source you want, including PuppetDB. From Puppet's perspective, the ENC submits a node name and gets back a hash of information.

External node classifiers can co-exist with standard node definitions in `site.pp`; the classes declared in each source are merged together.

### Merging classes from multiple sources

Every node always gets a node object from the configured node terminus. The node object might be empty, or it might contain classes, parameters, and an environment. The node terminus setting, `node_terminus`, takes effect where the catalog is compiled, on Puppet Server when using an agent-master configuration, and on the node itself when using `puppet apply`. The default node terminus is `plain`, which returns an empty node object, leaving node configuration to the main manifest. The `exec` terminus calls an ENC script to determine what should go in the node object. Every node might also get a node definition from the main manifest.

When compiling a node's catalog, Puppet includes all of the following:

- Classes specified in the node object it received from the node terminus.
- Classes or resources that are in the site manifest but outside any node definitions.
- Classes or resources in the most specific node definition in `site.pp` that matches the current node (if `site.pp` contains any node definitions). The following notes apply:
  - If `site.pp` contains at least one node definition, it must have a node definition that matches the current node; compilation fails if a match can't be found.
  - If the node name resembles a dot-separated fully qualified domain name, Puppet makes multiple attempts to match a node definition, removing the right-most part of the name each time. Thus, Puppet would first try `agent1.example.com`, then `agent1.example`, then `agent1`. This behavior isn't mimicked when calling an ENC, which is invoked only once with the agent's full node name.
  - If no matching node definition can be found with the node's name, Puppet tries one last time with a node name of `default`; most users include a `node default {}` statement in their `site.pp` file. This behavior isn't mimicked when calling an ENC.

## Comparing ENCs and node definitions

If you're trying to decide whether to use an ENC or main manifest node definitions (or both), consider the following:

- The YAML returned by an ENC isn't an exact equivalent of a node definition in `site.pp` — it can't declare individual resources, declare relationships, or do conditional logic. An ENC can only declare classes, assign top-scope variables, and set an environment. So, an ENC is most effective if you've done a good job of separating your configurations out into classes and modules.
- ENCs can set an environment for a node, overriding whatever environment the node requested.
- Even if you aren't using node definitions, you can still use `site.pp` to do things like set global resource defaults.
- Unlike regular node definitions, where a node can match a less specific definition if an exactly matching definition isn't found (depending on Puppet's `strict_hostname_checking` setting), an ENC is called only once, with the node's full name.

## Connect an ENC

Configure two settings to have Puppet Server connect to an external node classifier.

In the master's `puppet.conf` file:

1. Set the `node_terminus` setting to `exec`.
2. Set the `external_nodes` setting to the path to the ENC executable.

For example:

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

## ENC output format

An ENC must return either nothing or a YAML hash to standard out. The hash must contain at least one of `classes` or `parameters`, or it can contain both. It can also optionally contain an `environment` key.

ENCs exit with an exit code of 0 when functioning normally, and can exit with a non-zero exit code if you want Puppet to behave as though the requested node was not found.

If an ENC returns nothing or exits with a non-zero exit code, the catalog compilation fails with a “could not find node” error, and the node is unable to retrieve configurations.

For information about the YAML format, see yaml.org.

### Classes

If present, the value of `classes` must be either an array of class names or a hash whose keys are class names. That is, the following are equivalent:

```
classes:
  - common
  - puppet
  - dns
  - ntp

classes:
  common:
  puppet:
  dns:
  ntp:
```

If you're specifying parameterized classes, use the hash key syntax, not the array syntax. The value for a parameterized class is a hash of the class’s parameters and values. Each value can be a string, number, array, or hash. Put string values in quotation marks, because YAML parsers sometimes treat certain unquoted strings (such as `on`) as Booleans. Non-parameterized classes can have empty values.

```
classes:
    common:
    puppet:
    ntp:
        ntpserver: 0.pool.ntp.org
    aptsetup:
        additional_apt_repos:
            - deb localrepo.example.com/ubuntu lucid production
            - deb localrepo.example.com/ubuntu lucid vendor
```

### Parameters

If present, the value of the `parameters` key must be a hash of valid variable names and associated values; these are exposed to the compiler as top-scope variables. Each value can be a string, number, array, or hash.

```
parameters:
    ntp_servers:
        - 0.pool.ntp.org
        - ntp.example.com
    mail_server: mail.example.com
    iburst: true
```

### Environment

If present, the value of `environment` must be a string representing the desired environment for this node. This is the only environment used by the node in its requests for catalogs and files.

```
environment: production
```

### Complete example

```
---
classes:
    common:
    puppet:
    ntp:
        ntpserver: 0.pool.ntp.org
    aptsetup:
        additional_apt_repos:
            - deb localrepo.example.com/ubuntu lucid production
            - deb localrepo.example.com/ubuntu lucid vendor
parameters:
    ntp_servers:
        - 0.pool.ntp.org
        - ntp.example.com
    mail_server: mail.example.com
    iburst: true
environment: production
```

# Misc. references (settings, functions, etc.)

# Man pages

## Core tools

## Occasionally useful

## Niche

# HTTP API

## Puppet v4 API

## CA v1 API

## Schemas (JSON)

These JSON files contain schemas for the various HTTP API objects

- catalog.json
- environments.json
- error.json
- facts.json
- file_metadata.json
- host.json
- json-meta-schema.json
- node.json
- report.json
- status.json

# Certificate authority and SSL

Puppet can use its built-in certificate authority (CA) and public key infrastructure (PKI) tools or use an existing external CA for all of its secure socket layer (SSL) communications.

Puppet uses certificates to verify the the identity of nodes. These certificates are issued by the certificate authority (CA) service of a Puppet master. When a node checks into the Puppet master for the first time, it requests a certificate.

The Puppet master examines this request, and if it seems safe, creates a certificate for the node. When the agent node picks up this certificate, it knows it can trust the Puppet master, and it can now identify itself later when requesting a catalog.

After installing the Puppet Server, before starting it for the first time, use the `puppetserver ca setup` command to create a default intermediate CA. For more complex use cases, see the Intermediate and External CA documentation.

**Note:** For backward compatibility, starting Puppet Server before running `puppetserver ca setup` creates the old single-cert CA. This configuration is not recommended, and users of Puppet 6 should use the setup command instead.

Puppet provides two command line tools for performing SSL tasks:

- `puppetserver ca` signs certificate requests and revokes certificates.
- `puppet ssl` performs agent-side tasks, such as submitting a certificate request or downloading a node certificate.

### What's changed in Puppet 6

Puppet 6 removes the `puppet cert` command and its associated certificate-related faces. In Puppet 6 you must use the new subcommands listed above instead.

Puppet 6 also introduces full support for intermediate CAs, the recommended architecture. This requires changes on both the server and the agent, so using it requires both the server and the agent to be updated to Puppet 6.

- Puppet Server CA commands on page 486

Puppet Server has a `puppetserver ca` command that performs certificate authority (CA) tasks like signing and revoking certificates. Most of its actions are performed by making HTTP requests to Puppet Server's CA API, specifically the `certificate_status` endpoint. You must have Puppet Server running in order to sign or revoke certs.

- Intermediate CA
- Autosigning certificate requests on page 488

Before Puppet agent nodes can retrieve their configuration catalogs, they require a signed certificate from the local Puppet certificate authority (CA). When using Puppet's built-in CA instead of an external CA, agents submit a certificate signing request (CSR) to the CA to retrieve a signed certificate after it's available.

- CSR attributes and certificate extensions on page 491

When Puppet agent nodes request their certificates, the certificate signing request (CSR) usually contains only their certname and the necessary cryptographic information. Agents can also embed additional data in their CSR, useful for policy-based autosigning and for adding new trusted facts.

- Regenerating certificates in a Puppet deployment on page 496

In some cases, you might need to regenerate the certificates and security credentials (private and public keys) that are generated by Puppet's built-in PKI systems.

- External CA on page 499

This information describes the supported and tested configurations for external CAs in this version of Puppet. If you have an external CA use case that isn't listed here, contact Puppet so we can learn more about it.

- External SSL termination with Puppet Server

## Puppet Server CA commands

Puppet Server has a `puppetserver ca` command that performs certificate authority (CA) tasks like signing and revoking certificates. Most of its actions are performed by making HTTP requests to Puppet Server's CA API, specifically the `certificate_status` endpoint. You must have Puppet Server running in order to sign or revoke certs.

### CA subcommands

The following actions are available as subcommands for the `puppetserver ca` command:

- `clean`: Revoke certificates and remove related files from the CA.
- `generate`: Create a certificate signed by the CA.
- `import`: Import the CA's key, certificates, and certificate revocation lists (CRLs).
- `list`: List all certificate requests.
- `revoke`: Revoke a certificate.
- `setup`: Generate a root and intermediate signing CA for Puppet Server.
- `sign`: Sign a certificate.

Syntax:

```
puppetserver ca <action> [options]
```

Most commands require a target to be specified with the `--certname` flag. For example:

```
puppetserver ca sign --certname cert.example.com
```

The target is a comma separated list of names that act on multiple certificates at one time. You can supply a custom configuration file with the `--config` option. Some actions have additional options. Run `puppetserver ca help` for details.

These commands are shipped as a gem alongside Puppet Server. It can be updated between releases to pick up improvements and bug fixes. To update the gem, run:

```
/opt/puppetlabs/puppet/bin/gem install -i /opt/puppetlabs/puppet/lib/ruby/
vendor_gems puppetserver-ca
```

### API authentication

For security, access to the `certificate_status` API endpoint that is issued to sign and revoke certificates is tightly restricted. In Puppet 6, the master's certificate is generated with a special extension which is whitelisted in the `auth.conf` entries for the `certificate_status` and `certificate_statuses` endpoint. This extension is reserved, and Puppet Server refuses to sign other CSRs requesting it, even with `allow-authorization-extensions` set to `true`. If you need a certificate with this extension, you can generate it offline by stopping Puppet Server and running `puppetserver ca generate --ca-client --certname <name>`. API authentication is required for regenerating the master cert. For details on cert regeneration, see Regenerating certificates in a Puppet deployment.

### Upgrading

The Puppet CA commands are available in Puppet 5, but to use them, you must update Puppet Server's `auth.conf` to include a rule allowing the master's certname to access the `certificate_status` and `certificate_statuses` endpoints. If you're upgrading from Puppet 5 to Puppet 6 and you are not regenerating your CA, you must whitelist the master's certname. See Puppet Server configuration files: auth.conf for details on how to use `auth.conf`.

The following example shows how to allow the CA commands to access the `certificate_status` endpoint:

```
{
  match-request: {
 path: "/puppet-ca/v1/certificate_status"
     type: path
     method: [get, put, delete]
   }
   allow: master.example.com
   sort-order: 500
   name: "puppetlabs cert status"
},
```

#### Signing certificates with subject alternative names or auth extensions

With the removal of puppet cert sign, Puppet Server's CA API can sign certificates with subject alternative names (SANs) or auth extensions, which was previously disallowed. This option is disabled by default for security reasons, but you can turn it on by setting allow-subject-alt-names or allow-authorization-extensions to true in the certificate-authority section of Puppet Server's config (usually located in ca.conf). After these have been configured, you can use puppetserver ca sign --certname <name> to sign certificates with these additions.

## Autosigning certificate requests

Before Puppet agent nodes can retrieve their configuration catalogs, they require a signed certificate from the local Puppet certificate authority (CA). When using Puppet's built-in CA instead of an external CA, agents submit a certificate signing request (CSR) to the CA to retrieve a signed certificate after it's available.

By default, these CSRs must be manually signed by an admin user, using either the puppetserver ca command or the **Node requests** page in the Puppet Enterprise console.

Alternatively, to speed up the process of bringing new agent nodes into the deployment, you can configure the CA to automatically sign certain CSRs.

**CAUTION:** Autosigning CSRs changes the nature of your deployment's security, and you should understand the implications before configuring it. Each type of autosigning has its own security impact.

### Disabling autosigning

By default, the autosign setting in the [master] section of the CA's puppet.conf file is set to $confdir/autosign.conf. The basic autosigning functionality is enabled upon installation.

Depending on your installation method, there might not be a whitelist at that location after the Puppet Server is running:

- Open source Puppet: autosign.conf doesn't exist by default.
- Monolithic Puppet Enterprise (PE) installations: All required services run on one server, and autosign.conf exists on the master, but by default it's empty because the master doesn't need to whitelist other servers.
- Split PE installations: Services like PuppetDB can run on different servers, the autosign.conf exists on the CA master and contains a whitelist of other required hosts.

If the autosign.conf file is empty or doesn't exist, the whitelist is effectively empty. The CA Puppet master doesn't autosign any certificates until the the autosign setting's path is configured, or until the default autosign.conf file is a non-executable whitelist file. This file must contain correctly formatted content or a custom policy executable that the Puppet user has permission to run.

To explicitly disable autosigning, set autosign = false in the [master] section of the CA Puppet master's puppet.conf. This disables CA autosigning even if the autosign.conf file or a custom policy executable exists.

For more information about the autosign setting in puppet.conf, see the configuration reference.

### Naïve autosigning

Naïve autosigning causes the CA to autosign all CSRs.

To enable naïve autosigning, set autosign = true in the [master] section of the CA Puppet master's puppet.conf.

**CAUTION:** For security reasons, never use naïve autosigning in a production deployment. Naïve autosigning is suitable only for temporary test deployments that are incapable of serving catalogs containing sensitive information.

## Basic autosigning (`autosign.conf`)

In basic autosigning, the CA uses a config file containing a whitelist of certificate names and domain name globs. When a CSR arrives, the requested certificate name is checked against the whitelist file. If the name is present, or covered by one of the domain name globs, the certificate is autosigned. If not, it's left for a manual review.

### Enabling basic autosigning

The `autosign.conf` whitelist file's location and contents are described in its documentation.

Puppet looks for `autosign.conf` at the path configured in the `[autosign setting]` within the `[master]` section of `puppet.conf`. The default path is `$confdir/autosign.conf`, and the default `confdir` path depends on your operating system. For more information, see the confdir documentation.

If the `autosign.conf` file pointed to by the `autosign` setting is a file that the Puppet user can execute, Puppet instead attempts to run it as a custom policy executable, even if it contains a valid `autosign.conf` whitelist.

**Note:** In open source Puppet, no `autosign.conf` file exists by default. In Puppet Enterprise, the file exists by default but might be empty. In both cases, the basic autosigning feature is technically enabled by default but doesn't autosign any certificates because the whitelist is effectively empty.

The CA Puppet master therefore doesn't autosign any certificates until the `autosign.conf` file contains a properly formatted whitelist or is a custom policy executable that the Puppet user has permission to run, or until the `autosign` setting is pointed at a whitelist file with properly formatted content or a custom policy executable that the Puppet user has permission to run.

### Security implications of basic autosigning

Basic autosigning is insecure because any host can provide any certname when requesting a certificate. Use it only when you fully trust any computer capable of connecting to the Puppet master.

With basic autosigning enabled, an attacker who guesses an unused certname allowed by `autosign.conf` can obtain a signed agent certificate from the Puppet master. The attacker could then obtain a configuration catalog, which can contain sensitive information depending on your deployment's Puppet code and node classification.

## Policy-based autosigning

In policy-based autosigning, the CA runs an external policy executable every time it receives a CSR. This executable examines the CSR and tells the CA whether the certificate is approved for autosigning. If the executable approves, the certificate is autosigned; if not, it's left for manual review.

### Enabling policy-based autosigning

To enable policy-based autosigning, set `autosign = <policy executable file>` in the `[master]` section of the CA Puppet master's `puppet.conf`.

The policy executable file must be executable by the same user as the Puppet master. If not, it is treated as a certname whitelist file.

### Custom policy executables

A custom policy executable can be written in any programming language; it just has to be executable in a *nix-like environment. The Puppet master passes it the certname of the request (as a command line argument) and the PEM-encoded CSR (on stdin), and expects a `0` (approved) or non-zero (rejected) exit code.

After it has the CSR, a policy executable can extract information from it and decide whether to approve the certificate for autosigning. This is useful when you are provisioning your nodes and are embedding additional information in the CSR.

If you aren't embedding additional data, the CSR contains only the node's certname and public key. This can still provide more flexibility and security than `autosign.conf`, as the executable can do things like query your provisioning system, CMDB, or cloud provider to make sure a node with that name was recently added.

### Security implications of policy-based autosigning

Depending on how you manage the information the policy executable is using, policy-based autosigning can be fast and extremely secure.

For example:

- If you embed a unique pre-shared key on each node you provision, and provide your policy executable with a database of these keys, your autosigning security is as good as your handling of the keys. As long as it's impractical for an attacker to acquire a PSK, it's impractical for them to acquire a signed certificate.
- If nodes running on a cloud service embed their instance UUIDs in their CSRs, and your executable queries the cloud provider's API to check that a node's UUID exists in your account, your autosigning security is as good as the security of the cloud provider's API. If an attacker can impersonate a legit user to the API and get a list of node UUIDs, or if they can create a rogue node in your account, they can acquire a signed certificate.

When designing your CSR data and signing policy, you must think things through carefully. As long as you can arrange reasonable end-to-end security for secret data on your nodes, you should be able to configure a secure autosigning system.

### Policy executable API

The API for policy executables is as follows.

| | |
|---|---|
| Run environment | • The executable runs one time for each incoming CSR.<br>• It is executed by the Puppet master process and runs as the same user as the Puppet master.<br>• The Puppet master process is blocked until the executable finishes running. We expect policy executables to finish in a timely fashion; if they do not, it's possible for them to tie up all available Puppet master threads and deny service to other agents. If an executable needs to perform network requests or other potentially expensive operations, the author is in charge of implementing any necessary timeouts, possibly bailing and exiting non-zero in the event of failure. |
| Arguments | • The executable must allow a single command line argument. This argument is the Subject CN (certname) of the incoming CSR.<br>• No other command line arguments should be provided.<br>• The Puppet master should never fail to provide this argument. |
| Stdin | • The executable receives the entirety of the incoming CSR on its stdin stream. The CSR is encoded in `pem` format.<br>• The stdin stream contains nothing but the complete CSR.<br>• The Puppet master should never fail to provide the CSR on stdin. |

| | |
|---|---|
| Exit status | • The executable must exit with a status of `0` if the certificate should be autosigned; it must exit with a non-zero status if it should not be autosigned.<br>• The Puppet master treats all non-zero exit statuses as equivalent. |
| Stdout and stderr | • Anything the executable emits on stdout or stderr is copied to the Puppet Server log output at the `debug` log level. Puppet otherwise ignores the executable's output; only the exit code is considered significant. |

# CSR attributes and certificate extensions

When Puppet agent nodes request their certificates, the certificate signing request (CSR) usually contains only their certname and the necessary cryptographic information. Agents can also embed additional data in their CSR, useful for policy-based autosigning and for adding new trusted facts.

Embedding additional data into CSRs is useful when:

- Large numbers of nodes are regularly created and destroyed as part of an elastic scaling system.
- You are willing to build custom tooling to make certificate autosigning more secure and useful.

It might also be useful in deployments where Puppet is used to deploy private keys or other sensitive information, and you want extra control over nodes that receive this data.

If your deployment doesn't match one of these descriptions, you might not need this feature.

## Timing: When data can be added to CSRs and certificates

When Puppet agent starts the process of requesting a catalog, it checks whether it has a valid signed certificate. If it does not, it generates a key pair, crafts a CSR, and submits it to the certificate authority (CA) Puppet master. For detailed information, see agent/master HTTPS traffic.

For practical purposes, a certificate is locked and immutable as soon as it is signed. For data to persist in the certificate, it has to be added to the CSR before the CA signs the certificate.

This means any desired extra data must be present before Puppet agent attempts to request its catalog for the first time.

Populate any extra data when provisioning the node. If you make an error, see the Troubleshooting section below for information about recovering from failed data embedding.

## Data location and format

Extra data for the CSR is read from the `csr_attributes.yaml` file in Puppet's `confdir`. The location of this file can be changed with the `csr_attributes` configuration setting.

The `csr_attributes.yaml` file must contain a YAML hash with one or both of the following keys:

- `custom_attributes`
- `extension_requests`

The value of each key must also be a hash, where:

- Each key is a valid object identifier (OID) — Puppet-specific OIDs can optionally be referenced by short name instead of by numeric ID.
- Each value is an object that can be cast to a string — numbers are allowed but arrays are not.

For information about how each hash is used and recommended OIDs for each hash, see the sections below.

## Custom attributes (transient CSR data)

Custom attributes are pieces of data that are embedded only in the CSR. The CA can use them when deciding whether to sign the certificate, but they are discarded after that and aren't transferred to the final certificate.

### Default behavior

The `puppetserver ca list` command doesn't display custom attributes for pending CSRs, and basic autosigning (autosign.conf) doesn't check them before signing.

### Configurable behavior

If you use policy-based autosigning your policy executable receives the complete CSR in PEM format. The executable can extract and inspect the custom attributes, and use them to decide whether to sign the certificate.

The simplest method is to embed a pre-shared key of some kind in the custom attributes. A policy executable can compare it to a list of known keys and autosign certificates for any pre-authorized nodes.

A more complex use might be to embed an instance-specific ID and write a policy executable that can check it against a list of your recently requested instances on a public cloud, like EC2 or GCE.

### Manually checking for custom attributes in CSRs

You can check for custom attributes by using OpenSSL to dump a CSR in `pem` format to text format, by running this command:

```
openssl req -noout -text -in <name>.pem
```

In the output, look for the `Attributes` section which appears below the `Subject Public Key Info` block:

```
Attributes:
    challengePassword        :342thbjkt82094y0uthhor289jnqthpc2290
```

### Recommended OIDs for attributes

Custom attributes can use any public or site-specific OID, with the exception of the OIDs used for core X.509 functionality. This means you can't re-use existing OIDs for things like subject alternative names.

One useful OID is the `challengePassword` attribute — `1.2.840.113549.1.9.7`. This is a rarely-used corner of X.509 that can easily be repurposed to hold a pre-shared key. The benefit of using this instead of an arbitrary OID is that it appears by name when using OpenSSL to dump the CSR to text; OIDs that `openssl req` can't recognize are displayed as numerical strings.

You can also use the Puppet-specific OIDs.

## Extension requests (permanent certificate data)

Extension requests are pieces of data that are transferred as extensions to the final certificate, when the CA signs the CSR. They persist as trusted, immutable data, that cannot be altered after the certificate is signed.

They can also be used by the CA when deciding whether or not to sign the certificate.

### Default behavior

When signing a certificate, Puppet's CA tools transfer any extension requests into the final certificate.

You can access certificate extensions in manifests as `$trusted["extensions"]["<EXTENSION OID>"]`.

Select OIDs in the ppRegCertExt and ppAuthCertExt ranges. See the Puppet-specific Registered IDs. By default, any other OIDs appear as plain dotted numbers, but you can use the `custom_trusted_oid_mapping.yaml` file to assign short names to any other OIDs you use at your site. If you do, those OIDs appear in `$trusted` as their short names, instead of their full numerical OID.

For more information about `$trusted`, see facts and special variables.

The visibility of extensions is limited:

- The `puppetserver ca list` command does not display custom attributes for any pending CSRs, and basic autosigning (`autosign.conf`) doesn't check them before signing. Either use policy-based autosigning or inspect CSRs manually with the `openssl` command (see below).

Puppet's authorization system (`auth.conf`) does not use certificate extensions, but Puppet Server's authorization system, which is based on `trapperkeeper-authorization`, can use extensions in the `ppAuthCertExt` OID range, and requires them for requests to write access rules.

### Configurable behavior

If you use policy-based autosigning, your policy executable receives the complete CSR in `pem` format. The executable can extract and inspect the extension requests, and use them when deciding whether to sign the certificate.

### Manually checking for extensions in CSRs and certificates

You can check for extension requests in a CSR by running the OpenSSL command to dump a CSR in `pem` format to text format:

```
openssl req -noout -text -in <name>.pem
```

In the output, look for a section called `Requested Extensions`, which appears below the `Subject Public Key Info` and `Attributes` blocks:

```
Requested Extensions:
    pp_uuid:
    .$ED803750-E3C7-44F5-BB08-41A04433FE2E
    1.3.6.1.4.1.34380.1.1.3:
    ..my_ami_image
    1.3.6.1.4.1.34380.1.1.4:
    .$342thbjkt82094y0uthhor289jnqthpc2290
```

**Note:** Every extension is preceded by any combination of two characters (`.$` and `..` in the example above) that contain ASN.1 encoding information. Because OpenSSL is unaware of Puppet's custom extensions OIDs, it's unable to properly display the values.

Any Puppet-specific OIDs (see below) appear as numeric strings when using OpenSSL.

You can check for extensions in a signed certificate by running `puppet cert print <name>`. In the output, look for the `X509v3 extensions` section. Any of the Puppet-specific registered OIDs appear as their descriptive names:

```
X509v3 extensions:
    Netscape Comment:
        Puppet Ruby/OpenSSL Internal Certificate
    X509v3 Subject Key Identifier:
        47:BC:D5:14:33:F2:ED:85:B9:52:FD:A2:EA:E4:CC:00:7F:7F:19:7E
    Puppet Node UUID:
        ED803750-E3C7-44F5-BB08-41A04433FE2E
    X509v3 Extended Key Usage: critical
        TLS Web Server Authentication, TLS Web Client Authentication
    X509v3 Basic Constraints: critical
        CA:FALSE
    Puppet Node Preshared Key:
        342thbjkt82094y0uthhor289jnqthpc2290
    X509v3 Key Usage: critical
        Digital Signature, Key Encipherment
    Puppet Node Image Name:
```

```
        my_ami_image
```

### Recommended OIDs for extensions

Extension request OIDs must be under the `ppRegCertExt` (`1.3.6.1.4.1.34380.1.1`), `ppPrivCertExt` (`1.3.6.1.4.1.34380.1.2`), or `ppAuthCertExt` (`1.3.6.1.4.1.34380.1.3`) OID arcs.

Puppet provides several registered OIDs (under `ppRegCertExt`) for the most common kinds of extension information, a private OID range (`ppPrivCertExt`) for site-specific extension information, and an OID range for safe authorization to Puppet Server (`ppAuthCertExt`).

There are several benefits to using the registered OIDs:

- You can reference them in the `csr_attributes.yaml` file with their short names instead of their numeric IDs.
- You can access them in `$trusted[extensions]` with their short names instead of their numeric IDs.
- When using Puppet tools to print certificate info, they appear using their descriptive names instead of their numeric IDs.

The private range is available for any information you want to embed into a certificate that isn't widely used already. It is completely unregulated, and its contents are expected to be different in every Puppet deployment.

You can use the custom_trusted_oid_mapping.yaml file to set short names for any private extension OIDs you use. Note that this enables only the short names in the `$trusted[extensions]` hash.

### Puppet-specific registered IDs

#### `ppRegCertExt`

The `ppRegCertExt` OID range contains the following OIDs:

| Numeric ID | Short name | Descriptive name |
|---|---|---|
| 1.3.6.1.4.1.34380.1.1.1 | `pp_uuid` | Puppet node UUID |
| 1.3.6.1.4.1.34380.1.1.2 | `pp_instance_id` | Puppet node instance ID |
| 1.3.6.1.4.1.34380.1.1.3 | `pp_image_name` | Puppet node image name |
| 1.3.6.1.4.1.34380.1.1.4 | `pp_preshared_key` | Puppet node preshared key |
| 1.3.6.1.4.1.34380.1.1.5 | `pp_cost_center` | Puppet node cost center name |
| 1.3.6.1.4.1.34380.1.1.6 | `pp_product` | Puppet node product name |
| 1.3.6.1.4.1.34380.1.1.7 | `pp_project` | Puppet node project name |
| 1.3.6.1.4.1.34380.1.1.8 | `pp_application` | Puppet node application name |
| 1.3.6.1.4.1.34380.1.1.9 | `pp_service` | Puppet node service name |
| 1.3.6.1.4.1.34380.1.1.10 | `pp_employee` | Puppet node employee name |
| 1.3.6.1.4.1.34380.1.1.11 | `pp_created_by` | Puppet node `created_by` tag |
| 1.3.6.1.4.1.34380.1.1.12 | `pp_environment` | Puppet node environment name |
| 1.3.6.1.4.1.34380.1.1.13 | `pp_role` | Puppet node role name |
| 1.3.6.1.4.1.34380.1.1.14 | `pp_software_version` | Puppet node software version |
| 1.3.6.1.4.1.34380.1.1.15 | `pp_department` | Puppet node department name |
| 1.3.6.1.4.1.34380.1.1.16 | `pp_cluster` | Puppet node cluster name |
| 1.3.6.1.4.1.34380.1.1.17 | `pp_provisioner` | Puppet node provisioner name |

| Numeric ID | Short name | Descriptive name |
|---|---|---|
| 1.3.6.1.4.1.34380.1.1.18 | `pp_region` | Puppet node region name |
| 1.3.6.1.4.1.34380.1.1.19 | `pp_datacenter` | Puppet node datacenter name |
| 1.3.6.1.4.1.34380.1.1.20 | `pp_zone` | Puppet node zone name |
| 1.3.6.1.4.1.34380.1.1.21 | `pp_network` | Puppet node network name |
| 1.3.6.1.4.1.34380.1.1.22 | `pp_securitypolicy` | Puppet node security policy name |
| 1.3.6.1.4.1.34380.1.1.23 | `pp_cloudplatform` | Puppet node cloud platform name |
| 1.3.6.1.4.1.34380.1.1.24 | `pp_apptier` | Puppet node application tier |
| 1.3.6.1.4.1.34380.1.1.25 | `pp_hostname` | Puppet node hostname |

#### `ppAuthCertExt`

The `ppAuthCertExt` OID range contains the following OIDs:

| Numeric ID | Short name | Descriptive name |
|---|---|---|
| 1.3.6.1.4.1.34380.1.3.1 | `pp_authorization` | Certificate extension authorization |
| 1.3.6.1.4.1.34380.1.3.13 | `pp_auth_role` | Puppet node role name for authorization |

## Cloud provider attributes and extensions population example

To populate the `csr_attributes.yaml` file when you provision a node, use an automated script such as cloud-init.

For example, when provisioning a new node from the AWS EC2 dashboard, enter the following script into the **Configure Instance Details —> Advanced Details** section:

```
#!/bin/sh
if [ ! -d /etc/puppetlabs/puppet ]; then
   mkdir /etc/puppetlabs/puppet
fi
cat > /etc/puppetlabs/puppet/csr_attributes.yaml << YAML
custom_attributes:
    1.2.840.113549.1.9.7: mySuperAwesomePassword
extension_requests:
    pp_instance_id: $(curl -s http://169.254.169.254/latest/meta-data/
instance-id)
    pp_image_name:  $(curl -s http://169.254.169.254/latest/meta-data/ami-
id)
YAML
```

This populates the attributes file with the AWS instance ID, image name, and a pre-shared key to use with policy-based autosigning.

## Troubleshooting

### Recovering from failed data embedding

When testing this feature for the first time, you might not embed the right information in a CSR, or certificate, and might want to start over for your test nodes. This is not really a problem after your provisioning system is changed to populate the data, but it can easily happen when doing things manually.

To start over, do the following.

On the test node:

- Turn off Puppet agent, if it's running.
- If using Puppet version 6.0.3 or greater, run `puppet ssl clean`. If not, delete the following files:
  - `$ssldir/certificate_requests/<name>.pem`
  - `$ssldir/certs/<name>.pem`

On the CA Puppet master:

- Check whether a signed certificate exists. Use `puppetserver ca list --all` to see the complete list. If it exists, revoke and delete it with `puppetserver ca clean --certname <name>`.

After you've done that, you can start over.

## Regenerating certificates in a Puppet deployment

In some cases, you might need to regenerate the certificates and security credentials (private and public keys) that are generated by Puppet's built-in PKI systems.

For example, you might have a Puppet master you need to move to a different network in your infrastructure, or you might have experienced a security vulnerability that makes existing credentials untrustworthy.

**Note:** There are other, more automated ways of doing this. We recommend using Bolt to regenerate certs when needed. See the Bolt documentation for more information. There is also a `puppetlabs-certregen` module but is not supported with Puppet Server 6.

**Important:** The information on this page describes the steps for regenerating certs in an open source Puppet deployment. If you use Puppet Enterprise do not use the information on this page, as it leaves you with an incomplete replacement and non-functional deployment. Instead, PE customers must refer to one of the following pages:

- Regenerating certificates in split PE deployments
- Regenerating certificates in monolithic PE deployments

| If your goal is to... | Do this... |
|---|---|
| Regenerate an agent's certificate | Clear and regenerate certs for Puppet agents |
| Fix a compromised or damaged certificate authority | Regenerate the CA and all certificates |
| Completely regenerate all Puppet deployment certificates | Regenerate the CA and all certificates |
| Add DNS alt-names or other certificate extensions to your existing Puppet master | Regenerate the agent certificate of your Puppet master and add DNS alt-names or other certificates |

### Regenerate the agent certificate of your Puppet master and add DNS alt-names or other certificate extensions

This option preserves the master agent relationship and lets you add DNS alt-names or certificate extensions to your existing master.

1. Revoke the Puppet master's certificate and clean the CA files pertaining to it. Note that the agents won't be able to connect to the master until all of the following steps are finished.

   ```
   puppetserver ca clean --certname <CERTNAME_OF_YOUR_MASTER>
   ```

2. Remove the agent-specific copy of the public key, private key, and certificate-signing request pertaining to the certificate:

   ```
   puppet ssl clean
   ```

3. Stop the Puppet master service:

```
puppet resource service puppetserver ensure=stopped
```

   **Note:** The CA and server run in the same master so this also stops the CA.

4. After you've stopped the master and CA service, create a certificate signed by the CA and add DNS alt names (comma separated):

```
puppetserver ca generate --certname <CERTNAME> --subject-alt-names <DNS
ALT NAMES> --ca-client
```

   **Note:** If you don't want to add DNS alt names to your master, omit the --subject-alt-names <DNS ALT NAMES> option from the command above.

5. Restart the Puppet master service:

```
puppet resource service puppetserver ensure=running
```

## Regenerate the CA and all certificates

**CAUTION:** This process destroys the certificate authority and all other certificates. It is meant for use in the event of a total compromise of your site, or some other unusual circumstance. If you want to preserve the master agent relationship, regenerate the agent certificate of your Puppet master. If you just need to replace a few agent certificates, clear and regenerate certs for Puppet agents.

### Step 1: Clear and regenerate certs on your Puppet master

On the Puppet master hosting the CA:

1. Back up the SSL directory, which is in `/etc/puppetlabs/puppet/ssl/`. If something goes wrong, you can restore this directory so your deployment can stay functional. However, if you needed to regenerate your certs for security reasons and couldn't, get some assistance as soon as possible so you can keep your site secure.
2. Stop the agent service:

```
sudo puppet resource service puppet ensure=stopped
```

3. Stop the master service.

   For Puppet Server, run:

```
sudo puppet resource service puppetserver ensure=stopped
```

4. Delete the SSL directory:

```
sudo rm -r /etc/puppetlabs/puppet/ssl
```

5. Regenerate the CA and master's cert:

```
sudo puppetserver ca setup
```

   You will see this message: `Notice: Signed certificate request for ca`.

6. Start the Puppet master service by running:

```
sudo puppet resource service puppetserver ensure=running
```

7. Start the Puppet agent service by running this command:

```
sudo puppet resource service puppet ensure=running
```

At this point:

- You have a new CA certificate and key.
- Your Puppet master has a certificate from the new CA, and it can field new certificate requests.
- The Puppet master rejects any requests for configuration catalogs from nodes that haven't replaced their certificates. At this point, it is all of them except itself.
- When using any extensions that rely on Puppet certificates, like PuppetDB, the Puppet master won't be able to communicate with them. Consequently, it might not be able to serve catalogs, even to agents that do have new certificates.

#### Step 2: Clear and regenerate certs for any extension

You might be using an extension, like PuppetDB or MCollective, to enhance Puppet. These extensions probably use certificates from Puppet's CA in order to communicate securely with the Puppet master. For each extension like this, you'll need to regenerate the certificates it uses.

Many tools have scripts or documentation to help you set up SSL, and you can often just re-run the setup instructions.

### PuppetDB

We recommend PuppetDB users first follow the instructions in Step 3: Clear and regenerate certs for agents, below, because PuppetDB re-uses Puppet agents' certificates. After that, restart the PuppetDB service. See Redo SSL setup after changing certificates for more information.

#### Step 3: Clear and regenerate certs for Puppet agents

To replace the certs on agents, you'll need to log into each agent node and do the following steps.

1. Stop the agent service. On *nix:

   ```
   sudo puppet resource service puppet ensure=stopped
   ```

   On Windows, with Administrator privileges:

   ```
   puppet resource service puppet ensure=stopped
   ```

2. Locate Puppet's SSL directory and delete its contents.

   The SSL directory can be determined by running `puppet config print ssldir --section agent`
3. Restart the agent service. On *nix:

   ```
   sudo puppet resource service puppet ensure=running
   ```

   On Windows, with Administrator privileges:

   ```
   puppet resource service puppet ensure=running
   ```

   When the agent starts, it generates keys and requests a new certificate from the CA master.
4. If you are not using autosigning, log in to the CA master server and sign each agent node's certificate request.

   To view pending requests, run:

   ```
   sudo puppetserver ca list
   ```

   To sign requests, run:

   ```
   sudo puppetserver ca sign --certname <NAME>
   ```

   After an agent node's new certificate is signed, it's retrieved within a few minutes and a Puppet run starts.

After you have regenerated all agents' certificates, everything will be fully functional under the new CA.

**Note:** You can achieve the same results by turning these steps into Bolt tasks or plans. See the Bolt documentation for more information.

## External CA

This information describes the supported and tested configurations for external CAs in this version of Puppet. If you have an external CA use case that isn't listed here, contact Puppet so we can learn more about it.

### Supported external CA configurations

This version of Puppet supports some external CA configurations, however not every possible configuration is supported.

We fully support the following setup options:

- Single CA which directly issues SSL certificates.
- Puppet Server functioning as an intermediate CA.

Fully supported by Puppet means:

- If issues arise that are considered bugs, we'll fix them as soon as possible.
- If issues arise in any other external CA setup that are considered feature requests, we'll consider whether to expand our support.

## Option 1: Puppet Server functioning as an intermediate CA

Puppet Server can operate as an intermediate CA to an external root CA.

See Using Puppet Server as an intermediate certificate authority.

## Option 2: Single CA

When Puppet uses its internal CA, it defaults to a single CA configuration. A single externally issued CA can also be used in a similar manner.

```
            +-----------------------+
            |                       |
            |  Root self-signed CA  |
            |                       |
            +------+----------+------+
                   |          |
        +----------+          +------------+
        |                                  |
        v                                  v
+----------------+                +----------------+
|                |                |                |
| Server SSL Cert |               | Agent SSL Cert |
|                |                |                |
+----------------+                +----------------+
```

This is an all or nothing configuration rather than a mix-and-match. When using an external CA, the built-in Puppet CA service must be disabled and cannot be used to issue SSL certificates.

**Note:** Puppet cannot automatically distribute certificates in this configuration.

### Puppet Server

Configure Puppet Server in three steps:

- Disable the internal CA service.
- Ensure that the certname does not change.
- Put certificates and keys in place on disk.

1. To disable the internal CA, edit the Puppet Server `/etc/puppetlabs/puppetserver/services.d/ca.cfg` file to comment and uncomment the appropriate settings:

```
# To enable the CA service, leave the following line uncommented
puppetlabs.services.ca.certificate-authority-service/certificate-authority-service
# To disable the CA service, comment out the above line and uncomment the
 line below
#puppetlabs.services.ca.certificate-authority-disabled-service/certificate-authority-disabled-service
puppetlabs.trapperkeeper.services.watcher.filesystem-watch-service/filesystem-watch-service
```

2. Set a static value for the `certname` setting in `puppet.conf`:

```
[master]
certname = puppetserver.example.com
```

   Setting a static value prevents any confusion if the machine's hostname changes. The value must match the certname you'll use to issue the server's certificate, and it must not be blank.

3. Put the credentials from your external CA on disk in the correct locations. These locations must match what's configured in your `webserver.conf` file.

   If you haven't changed those settings, run the following commands to find the default locations.

| Credential | File location |
|---|---|
| Server SSL certificate | `puppet config print hostcert --section master` |
| Server SSL certificate private key | `puppet config print hostprivkey --section master` |
| Root CA certificate | `puppet config print localcacert --section master` |
| Root certificate revocation list | `puppet config print hostcrl --section master` |

   If you've put the credentials in the correct locations, you don't need to change any additional settings.

### Puppet agent

You don't need to change any settings. Put the external credentials into the correct filesystem locations. You can run the following commands to find the appropriate locations.

| Credential | File location |
|---|---|
| Agent SSL certificate | `puppet config print hostcert --section agent` |
| Agent SSL certificate private key | `puppet config print hostprivkey --section agent` |
| Root CA certificate | `puppet config print localcacert --section agent` |
| Root certificate revocation list | `puppet config print hostcrl --section agent` |

## General notes and requirements

### PEM encoding of credentials is mandatory

Puppet expects its SSL credentials to be in `.pem` format.

### Normal Puppet certificate requirements still apply

Any Puppet Server certificate must contain the DNS name, either as the Subject Common Name (CN) or as a Subject Alternative Name (SAN), that agent nodes use to attempt contact with the server.

### Client DN authentication

Puppet Server is hosted by a Jetty web server; therefore. For client authentication purposes, Puppet Server can extract the distinguished name (DN) from a client certificate provided during SSL negotiation with the Jetty web server.

The use of an `X-Client-DN` request header is supported for cases where SSL termination of client requests needs to be done on an external server. See External SSL Termination with Puppet Server for details.

### Web server configuration

Use the `webserver.conf` file for Puppet Server to configure Jetty. Several `ssl-` settings can be added to the webserver.conf file to enable the web server to use the correct SSL configuration:

- `ssl-cert`: The value of `puppet master --configprint hostcert`. Equivalent to the 'SSLCertificateFile' Apache config setting.
- `ssl-key`: The value of `puppet master --configprint hostprivkey`. Equivalent to the 'SSLCertificateKeyFile' Apache config setting.
- `ssl-ca-cert`: The value of `puppet master --configprint localcacert`. Equivalent to the 'SSLCACertificateFile' Apache config setting.
- `ssl-cert-chain`: Equivalent to the 'SSLCertificateChainFile' Apache config setting. Optional.
- `ssl-crl-path`: The path to the CRL file to use. Optional.

An example `webserver.conf` file might look something like this:

```
webserver: {
 client-auth : want
 ssl-host    : 0.0.0.0
 ssl-port    : 8140
 ssl-cert    : /path/to/master.pem
 ssl-key     : /path/to/master.key
 ssl-ca-cert : /path/to/ca_bundle.pem
 ssl-cert-chain : /path/to/ca_bundle.pem
 ssl-crl-path : /etc/puppetlabs/puppet/ssl/crl.pem
}
```

For more information on these settings, seeConfiguring the Web Server Service.

### Restart required

After the above changes are made to Puppet Server's configuration files, you'll have to restart the Puppet Server service for the new settings to take effect.

# Puppet's internals

Learn the details of Puppet's internals, including how masters and agents communicate via host-verified HTTPS, and about the process of catalog compilation.

• Agent-master HTTPS communications on page 502
The Puppet agent and master communicate via mutually authenticated HTTPS using client certificates.
• Catalog compilation on page 504
When configuring a node, the agent uses a document called a catalog, which it downloads from the master. For each resource under management, the catalog describes its desired state and can specify ordered dependency information.

## Agent-master HTTPS communications

The Puppet agent and master communicate via mutually authenticated HTTPS using client certificates.

The HTTPS endpoints that Puppet uses are documented in the HTTP API reference. Access to each endpoint is controlled by `auth.conf` settings. For more information, see Puppet Server configuration files: auth.conf.

### Persistent HTTP and HTTPS connections and Keep-Alive

When acting as an HTTPS client, Puppet reuses connections by sending `Connection: Keep-Alive` in HTTP requests. This reduces transport layer security (TLS) overhead, improving performance for runs with dozens of HTTPS requests.

You can configure the `Keep-Alive` duration using the `http_keepalive_timeout` setting, but it must be shorter than the maximum `keepalive` allowed by the master's web server.

Puppet caches HTTP connections and verified HTTPS connections. If you specify a custom HTTP connection class, Puppet does not cache the connection.

Puppet always requests that a connection is kept open, but the server can choose to close the connection by sending `Connection: close` in the HTTP response. If that occurs, Puppet does not cache the connection and starts a new connection for its next request.

For more information about the `http_keepalive_timeout` setting, see the Configuration reference.

For an example of a server disabling persistent connections, see the Apache documentation on KeepAlive.

**The process of Agent-side checks and HTTPS requests during a single Puppet run.**

1. Check for keys and certificates:

   a. The agent downloads the CA (Certification Authority) bundle.
   b. If certificate revocation is enabled, the agent loads or downloads the Certificate Revocation List (CRL) bundle using the previous CA bundle to verify the connection.
   c. The agent loads or generates a private key. If the agent needs a certificate, it generates a Certificate Signing Request (CSR), including any `dns_alt_names` and `csr_attributes`, and submits the request using `PUT /puppet-ca/v1/certificate_request/:certname`.
   d. The agent attempts to download the signed certificate using `GET /puppet-ca/v1/certificate/:certname`.

      - If there is a conflict that must be resolved on the Puppet server, such as cleaning the old CSR or certificate, the agent sleeps for `waitforcert` seconds, or exits with `1` if waiting is not allowed, such as when running `puppet agent -t`.

        **Tip:** This can happen if the agent's SSL directory is deleted, as the Puppet server still has the valid, unrevoked certificate.
      - If the downloaded certificate fails verification, such as it does not match its private key, then Puppet discards the certificate. The agent sleeps for `waitforcert` seconds, or exits with `1` if waiting is not allowed, such as when running `puppet agent -t`.
2. Request a node object and switch environments:

   - Do a GET request to `/puppet/v3/node/<NAME>` .

     - If the request is successful, read the environment from the node object. If the node object has an environment, use that environment instead of the one in the agent's config file in all subsequent requests during this run.
     - If the request is unsuccessful, or if the node object had no environment set, use the environment from the agent's config file.
3. If `pluginsync` is enabled on the agent, fetch plugins from a file server mountpoint that scans the `lib` directory of every module:

   - Do a GET request to `/puppet/v3/file_metadatas/plugins` with `recurse=true` and `links=manage`.
   - Check whether any of the discovered plugins need to be downloaded. If so, do a GET request to `/puppet/v3/file_content/plugins/<FILE>` for each one.
4. Request catalog while submitting facts:

   - Do a POST request to `/puppet/v3/catalog/<NAME>`, where the post data is all of the node's facts encoded as JSON. Receive a compiled catalog in return.

     **Note:** Submitting facts isn't logically bound to requesting a catalog. For more information about facts, see Language: Facts and built-in variables.

5. Make file source requests while applying the catalog:

   File resources can specify file contents as either a `content` or `source` attribute. Content attributes go into the catalog, and the agent needs no additional data. Source attributes put only references into the catalog and might require additional HTTPS requests.

   - If you are using the normal compiler, then for each file source, the agent makes a GET request to `/puppet/v3/file_metadata/<SOMETHING>` and compares the metadata to the state of the file on disk.
     - If it is in sync, it continues on to the next file source.
     - If it is out of sync, it does a GET request to `/puppet/v3/file_content/<SOMETHING>` for the content.
   - If you are using the static compiler, all file metadata is embedded in the catalog. For each file source, the agent compares the embedded metadata to the state of the file on disk.
     - If it is in sync, it continues on to the next file source.
     - If it is out of sync, it does a GET request to `/puppet/v3/file_bucket_file/md5/<CHECKSUM>` for the content.

       **Note:** Using a static compiler is more efficient with network traffic than using the normal (dynamic) compiler. Using the dynamic compiler is less efficient during catalog compilation. Large amounts of files, especially recursive directories, amplifies either issue.
6. If `report` is enabled on the agent, submit the report:
   - Do a PUT request to `/puppet/v3/report/<NAME>`. The content of the PUT should be a Puppet report object in YAML format.

## Catalog compilation

When configuring a node, the agent uses a document called a catalog, which it downloads from the master. For each resource under management, the catalog describes its desired state and can specify ordered dependency information.

Puppet manifests are concise because they can express variation between nodes with conditional logic, templates, and functions. Puppet resolves these on the master and gives the agent a specific catalog.

This allows Puppet to:

- Separate privileges, because each node receives only its own resources.
- Reduce the agent’s CPU and memory consumption.
- Simulate changes by running the agent in no-op mode, checking the agent's current state and reporting what would have changed without making any changes.
- Query PuppetDB for information about managed resources on any node.

**Note:** The `puppet apply` command compiles the catalog on its own node and then applies it, so it plays the role of both master and agent. To compile a catalog on the master for testing, run `puppet catalog compile` on the `puppetserver` with access to your environments, modules, manifests, and Hiera data.

For more information about PuppetDB queries, see PuppetDB API.

Puppet compiles a catalog using three sources of configuration information:

- Agent-provided data
- External data
- Manifests and modules, including associated templates and file sources

These sources are used by both agent-master deployments and by stand-alone Puppet apply nodes.

### Agent-provided data

When an agent requests a catalog, it sends four pieces of information to the master:

- The node's name, which is almost always the same as the node's certname and is embedded in the request URL. For example, `/puppet/v3/catalog/web01.example.com?environment=production`.
- The node's certificate, which contains its certname and sometimes additional information that can be used for policy-based autosigning and adding new trusted facts. This is the one item not used by Puppet apply.
- The node's facts.
- The node's requested environment, which is embedded in the request URL. For example, `/puppet/v3/catalog/web01.example.com?environment=production`. Before requesting a catalog, the agent requests its environment from the master. If the master doesn't provide an environment, the environment information in the agent's config file is used.

For more information about additional data in certs see SSL configuration: CSR attributes and certificate extensions

### External data

Puppet uses two main kinds of external data during catalog compilation:

- Data from an external node classifier (ENC) or other node terminus, which is available before compilation starts. This data is in the form of a node object and can contain any of the following:
  - Classes
  - Class configuration parameters
  - Top-scope variables for the node
  - Environment information, which overrides the environment information in the agent's configuration
- Data from other sources, which can be invoked by the main manifest or by classes or defined types in modules. This kind of data includes:
  - Exported resources queried from PuppetDB.
  - The results of functions, which can access data sources including Hiera or an external configuration management database.

For more information about ENCs, see Writing external node classifiers

### Manifests and modules

Manifests and modules are at the center of a Puppet deployment, including the main manifest, modules downloaded from the Forge , and modules written specifically for your site.

For more information about manifests and modules, see The main manifest directory and Module fundamentals.

### The catalog compilation process

This simplified description doesn't delve into the internals of the parser, model, and the evaluator. Some items are presented out of order for the sake of clarity. This process begins after the catalog request has been received.

**Note:** For practical purposes, treat `puppet apply` nodes as a combined agent and master.

1. Retrieve the node object.
   - After the master has received the agent-provided information for this request, it asks its configured node terminus for a node object.
   - By default, the master uses the `plain` node terminus, which returns a blank node object. In this case, only manifests and agent-provided information are used in compilation.
   - The next most common node terminus is the `exec` node terminus, which requests data from an ENC. This can return classes, variables, an environment, or a combination of the three, depending on how the ENC is designed.
   - You can also write a custom node terminus that retrieves classes, variables, and environments from an external system.

2. Set variables from the node object, from facts, and from the certificate.
    - All of these variables are available for use by any manifest or template during subsequent stages of compilation.
    - The node’s facts are set as top-scope variables.
    - The node’s facts are set in the protected `$facts` hash, and certain data from the node’s certificate is set in the protected `$trusted` hash.
    - Any variables provided by the master are set.
3. Evaluate the main manifest.
    - Puppet parses the main manifest. The node’s environment can specify a main manifest; if it doesn’t, the master uses the main manifest from the agent's config file.
    - If there are node definitions in the manifest, Puppet must find one that matches the node’s name. If at least one node definition is present and Puppet cannot find a match, it fails compilation.
    - Code outside of node definitions is evaluated. Resources in the code are added to the are added to the node's catalog, and any classes declared in the code are loaded and declared.

      **Note:** Classes are usually classes are defined in modules, although the main manifest can also contain class definitions.
    - If a matching node definition is found, the code in it is evaluated at node scope, overriding any top-scope variables. Resources in the code are added to the are added to the node's catalog, and any classes declared in the code are loaded and declared.
4. Load and evaluate classes from modules
    - If classes were declared in the main manifest and their definitions were not present, Puppet loads the manifests containing them from its collection of modules. It follows the normal manifest naming conventions to find the files it should load. The set of locations Puppet loads modules from is called the modulepath. The master serves each environment with its own modulepath. When a class is loaded, the Puppet code in it is evaluated, and any resources in it are added to the catalog. If it was declared at node scope, it has access to node-scope variables; otherwise, it has access to only top-scope variables. Classes can also declare other classes; if they do, Puppet loads and evaluates those in the same way.
5. Evaluate classes from the node object
    - Puppet loads from modules and evaluate any classes that were specified by the node object. Resources from those classes are added to the catalog. If a matching node definition was found when the main manifest was evaluated, these classes are evaluated at node scope, which means that they can access any node-scope variables set by the main manifest. If no node definitions were present in the main manifest, they are evaluated at top scope.

### Related information

Writing external node classifiers on page 481
An external node classifier (ENC) is a script or application that tells Puppet which classes a node should have. It can replace or work in concert with the node definitions in the main site manifest (`site.pp`).

Scope on page 369
A scope is a specific area of code that is partially isolated from other areas of code.

Node definitions on page 252
A node definition, also known as a node statement, is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.

The modulepath on page 86
The master service and the `puppet apply` command load most of their content from modules found in one or more directories. The list of directories where Puppet looks for modules is called the *modulepath*. The modulepath is set by the current node's environment.

Exported resources on page 362

An exported resource declaration specifies a desired state for a resource, and publishes the resource for use by other nodes. It does not manage the resource on the target system. Any node, including the node that exports it, can collect the exported resource and manage its own copy of it.

# Experimental features

Released versions of Puppet can include experimental features to be considered for adoption but are not yet ready for production. These features need to be tested in the field before they can be considered safe, and therefore are turned off by default.

Experimental features can have a solid design but with an unknown performance and resource usage. Sometimes even the design is tentative, and because of this, we need feedback from users. By shipping these features early in disabled form, we want it to be easier for testing and giving feedback.

### Risks and support

**CAUTION:** Experimental features are not officially supported by Puppet, and we do not recommend that you turn them on in a production environment. They are available for testing in relatively safe scratch environments, and are used at your own risk.

Puppet employees and community members do their best to help you in informal channels like IRC, and the puppet-users and puppet-dev mailing lists, but we make no promises about experimental functionality.

Enabling experimental features might degrade the performance of your Puppet infrastructure, interfere with the normal operation of your managed nodes, introduce unexpected security risks, or have other undesired effects.

This is especially relevant to Puppet Enterprise customers. If Puppet support is assisting you with a problem, we might ask you to disable any experimental features.

### Changes to experimental features

Experimental features are exempt from semantic versioning, which means that they can change at any time, and are not limited to major or minor release boundaries.

These changes might include adding or removing functionality, changing the names of settings and other affordances, and more.

### Documentation of experimental features

The Puppet documentation contains pages for currently available experimental features. These pages are focused on enabling a feature and running through the interesting parts of its functionality; they might lag slightly behind the feature as implemented.

When a feature has experienced major changes across minor versions, we note the differences at the top of that feature page.

Each feature page attempts to give some context about the status of that feature and its prospects for official release.

### Giving feedback on experimental features

To help us keep improving Puppet, tell us more about your experience.

The best places to talk about experimental features are the puppet-users and puppet-dev mailing lists. This tells us what's working and what isn't, while also helping others learn from your experience. For more information about the Puppet mailing lists, see the community guidelines for mailing lists.

For more immediate conversations, you can use the #puppet and #puppet-dev channels on irc.freenode.net. For more information about these channels, see the community guidelines for IRC.

- Msgpack support on page 508

Puppet agents and masters communicate over HTTPS, exchanging structured data in JSON, or PSON which allows binary data.

## Msgpack support

Puppet agents and masters communicate over HTTPS, exchanging structured data in JSON, or PSON which allows binary data.

Msgpack is an efficient serialization protocol that behaves similarly to JSON. It provides faster and more robust serialization for agent-master communications, without requiring many changes in our code.

**Important:** When msgpack is enabled, the Puppet master and agent communicates using msgpack instead of PSON.

### Enabling Msgpack serialization

Enabling msgpack is easy, but first, it must be installed because the gem is not included in the puppet-agent or puppetserver packages.

1. Install the msgpack gem on your master and all agent nodes.

   If you are using the Puppet Enterprise test environment, make sure to use PE gem command instead of the system gem command.

   On *nix nodes, run the following command:

   ```
   /opt/puppetlabs/puppet/bin/gem install msgpack
   ```

   On Windows nodes, run the following command:

   ```
   "C:\Program Files\Puppet Labs\Puppet\sys\ruby\bin\gem" install msgpack
   ```

   On Puppet Server, run the following command and then restart the Puppet Server service:

   ```
   puppetserver gem install msgpack
   ```
2. In the `[agent]` or `[main]` section of `puppet.conf` on any number of agent nodes, set the `preferred_serialization_format` setting to `msgpack`.

After this is configured, the Puppet master server uses msgpack when serving any agents that have `preferred_serialization_format` set to `msgpack`. Any agents without that setting continue to receive PSON as normal.