Open source Puppet documentation

Contents

Welcome to Puppet documentation

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 Hiera Facter Puppet Server PuppetDB **Installing and upgrading** Release notes System requirements Pre-install instructions Install agents: Linux, Windows, and macOS Configuring Upgrading from Puppet 4 or later Upgrading from Puppet 3.8.x **The details** Resource type reference and cheat sheet Language summary Language visual index Language style guide **Modules** Fundamentals Beginner's guide Puppet Strings Puppet Strings style guide	**Other Puppet open source docs and projects** Puppet Development Kit docs Bolt docs Open source projects on Github **Get involved** Forge Community **Learn more about Puppet** Blog posts about open source Puppet Puppet training Learning roadmap **Need more?** Try Puppet Enterprise Puppet Enterprise docs Continuous Delivery for Puppet Enterprise docs **Select Puppet docs in other languages** Japanese (###) Spanish (Español) German (Deutsch)

To send us feedback or let us know about a docs error, open a ticket (you'll need a Jira account) or email the docs team.

Puppet 5 Platform

Quick start guides

Deprecated features

Installing and upgrading

Install: Puppet agent

Major upgrade from Puppet 3.8.x

Configuration

Config files

Configuring Puppet Server

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, a Windows user with administrator privileges, or 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`, you should usually 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`.

Note: Running the master as a Rack application is deprecated. When Puppet master is running as a Rack application, the `config.ru` file must explicitly set `--codedir` to the system codedir. The example `config.ru` file provided with the Puppet source does this.

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, Hiera data, and Hiera's configuration file, `hiera.yaml`.

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 config 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, such as `puppet cert`, with `sudo`.

Note: When Puppet master is running as a Rack application, the `config.ru` file must explicitly set `--confdir` to the system confdir. The example `config.ru` file provided with the Puppet source does this.

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 will use 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` or `puppet cert` 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 Puppet versions 4.0 to 4.4, if a `hiera.yaml` file exists in the global codedir, it takes precedence over the `hiera.yaml` in the global confdir. For Puppet to honor the `hiera.yaml` in the confdir, there must be no `hiera.yaml` file 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, it will use a system-wide cache directory. When running as a non-root user, it will use a cache directory in the user's home directory.

Since you will 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 will use 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`	

Directory name	Config setting	Notes
`client_data`	`client_datadir`	
`clientbucket`	`clientbucketdir`	
`client_yaml`	`clientyamldir`	
`devices`	`devicedir`	
`lib/facter`	`factpath`	
`facts`	`factpath`	
`facts.d`	`pluginfactdest`	
`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`	

File or directory name	Config setting	Notes
`state.yaml`	`statefile`	

Environments

Environments are isolated groups of agent nodes.

- About environments on page 18

An environment is a branch that gets turned into a directory on your master.

- Creating environments on page 19

An environment is a branch that gets turned into a directory on your Puppet master. Environments are turned on by default.

- Environment isolation on page 23

Environment isolation prevents resource types from leaking between your various environments.

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 will be 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. Once 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 will be 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 will use when compiling catalogs for agent nodes. The `modulepath`, the main manifest, Hiera data, and the config version script, can all be specified in `envronment.conf`.

The modulepath

The `modulepath` is the list of directories will load modules from. By default, will load 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 will only use 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 will use 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 will 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 will 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` will 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 will be 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 will 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` will not be 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`, since 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 will determine 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` will not be 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 will look 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 will be 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 will use 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 will request that environment. If you are using an ENC and it specifies an environment for that node, it will override 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 will fail to compile its

catalog. The one exception to this is if the default production environment does not exist. In this case, the agent will successfully retrieve 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 will look 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` will be 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. Once your code deployment process is mature, change this setting to unlimited.

All code in Ruby and Puppet loaded from the environment is cached. Inputs to compilation (for example, facts and looked up values) and the resulting catalog, are not cached.

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 will use the same global manifest for every environment. If an environment specifies a different manifest in `environment.conf`, Puppet will 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 will 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 once 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 will 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. 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 Plug-ins in modules Modules cheatsheet **Managing modules** Installing modules Upgrading modules Uninstalling modules `puppet-module` command reference **Writing modules** Beginner's guide to modules Module metadata Publishing modules **Documenting modules** Writing module documentation Puppet Strings Puppet Strings style guide **Publishing modules** On the Forge **Contributing to modules** Contributing to Puppet modules Reviewing community pull requests	**Modules on the Puppet Forge** The Forge module repository Puppet approved modules Puppet supported modules About Forge module quality scoring **Modules in Code Manager** Managing modules with the Puppetfile **Puppet Language reference** Classes Defined types Puppet tasks and plans **Developing and testing modules** Puppet Development Kit (PDK) **Developing types and providers** Puppet Resource API **Community resources** Puppet Community Slack 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.

`init.pp`: The main class of the module.

`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.

`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.

`facter/`

Contains custom facts, written in Ruby.

`puppet/`

Contains custom functions, resource types, and resource providers:

`puppet/functions/`: Contains functions written in Ruby for the modern `Puppet::Functions` API.

`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.

`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.

`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.

`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`.

`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.

`implementation/my_defined_type.pp`: This defined type is named `my_module::implementation::my_defined_type`.

`implementation/class.pp`: This defined type is named `my_module::implementation::class`.

`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.

`component.erb`

A manifest can render this template with `template('my_module/component.erb')`.

`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`
`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 will return 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.

The `puppet module generate` command is deprecated and will be removed in a future version of Puppet. 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 41
Create great Puppet modules by following best practices and guidelines.

Documenting modules on page 51
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 96
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`
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 25
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 29
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, but you must have a valid PE license on the machine on which you download the module.

To install or upgrade a PE-only module with the `puppet module` command, you must:

- Be logged in as the root user.
- Install the module on a PE-licensed node.
- Have internet access on the machine you are using to download modules.

Note: If you use `librarian-puppet` to manage PE modules, you must first install the module, and then commit the module to your version control repository.

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.

Before you begin

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 since 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`

Action	Description	Arguments	Example
`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`
`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	None.	If not specified, `puppet module install` exits and returns information if it encounters installation errors or conflicts.

Option	Description	Value	Default
	directory. Does not install dependencies.		
`--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.
`--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.

Option	Description	Value	Default
`--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.
`--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.

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.

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

When you author a module, it must contain certain metadata in a `metadata.json` file, which contains important information that Puppet, the Forge, and your module's users rely on.

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" }
]
```

When installing modules with the `puppet module install` command, Puppet installs any missing dependencies. When installing modules with Code Manager and the Puppetfile, dependencies are not automatically installed, so they must be explicitly specified in the 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.

For example, this key specifies that the module works with any Puppet version of 5.5.0 or greater, but not with Puppet 6 or later:

```
"requirements": [
  {"name": "puppet”, “version_requirement”: ">= 5.5.0 < 6.0.0"}
]
```

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 5.5.0 and no upper bound, the Forge applies an upper bound of `< 6.0.0`.

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.

Format	Description
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	`"Apache-2.0"`

Key	Required?	Value	Example
		a complete list, see the SPDX license list.	
`"summary"`	Required.	A one-line description of your module.	`"Standard library of resources for Puppet modules."`
`"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" ]`

Key	Required?	Value	Example
			`}`
`"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."
- 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 56
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 62
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. Once 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 requiremements

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" may be 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 will overwrite 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 51
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 62
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.

Before you begin

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/foo/lib/**/*.rb' 'modules/foo/
manifests/**/*.pp' 'modules/foo/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: • HTML: `./doc/` • Markdown: main module directory) • 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" • "textile" • "rdoc" • "ruby" • "text" • "html" • "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" • "textile" • "rdoc" • "ruby" • "text" • "html" • "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 will be 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 will be 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 once.
# @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 param.
# @param param2
#   Second param.
# @param param3
#   Third param.
plan baz(String $param1, $param2, Integer $param3 = 1) {
  run_task('foo::bar', $param1, blerg => $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 may not 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 46
When you author a module, it must contain certain metadata in a `metadata.json` file, which contains important information that Puppet, the Forge, and your module's users rely on.

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 will be 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 will contain 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 will receive the following error:

```
Warning: Symlinks in modules are unsupported. Please investigate symlink
 manifests/foo.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 will load the new release page for your module. If there were any errors on your upload, they will 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 will build 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 will be 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 will be visible to Forge users.
6. Click **Yes, delete it**.

On your module page, you will see a banner confirmation of the deletion.

Related information

Finding and downloading deleted modules on page 33
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 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. It provides the same services as the classic Puppet master application, and more.

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 vs. Apache/Passenger master
- Puppet Server Subcommands
- Puppet Master's Man Page

Classic Rack-based Puppet master

Important: Before Puppet Server was released, most users ran a Rack-based Puppet master using the Apache + Passenger stack. This still works, but it's deprecated and will be removed in Puppet 5.

Using Puppet modules and various other data sources, a Rack-based Puppet master compiles and serves configuration catalogs for any number of Puppet agents.

CAUTION: It's possible to run a standalone WEBrick-based Puppet master, however we don't recommend doing this in production.

For more information, see:

- Overview of Puppet's Architecture
- The Rack Puppet Master
- The WEBrick Puppet Master
- 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 cert

Puppet cert is an administrative tool that helps manage Puppet's built-in certificate authority (CA). It runs on the same server as the master, and you can use it to sign and revoke agent certificates. For more information, see Puppet Cert's Man Page.

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. Since 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 will 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`.

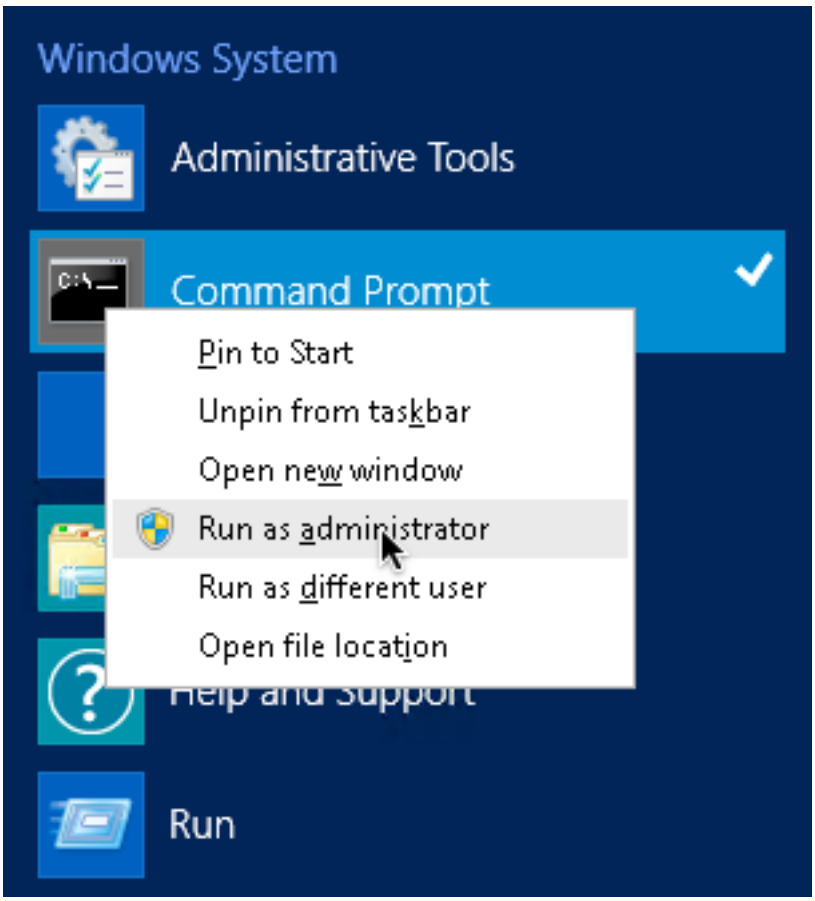
On *nix systems, Puppet defaults to running with limited privileges, when not run by `root`, but can have its privileges raised with the standard sudo command.

Windows systems don't use sudo, so escalating privileges works differently.

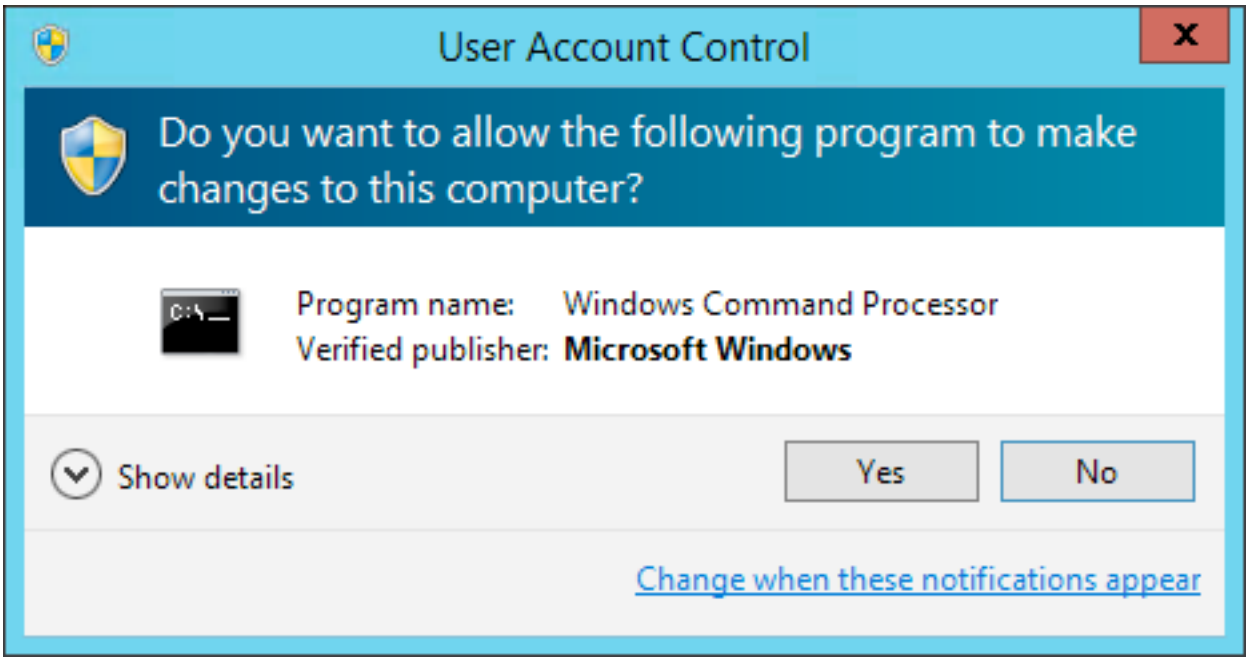
Newer versions of Windows manage security with User Account Control (UAC), which was added in Windows 2008 and Windows Vista. With UAC, most programs run by administrators will still have limited privileges. To get administrator privileges, the process has to request those privileges when it starts.

To run Puppet's commands in adminstrator mode, you must first start a Powershell command prompt with administrator privileges.

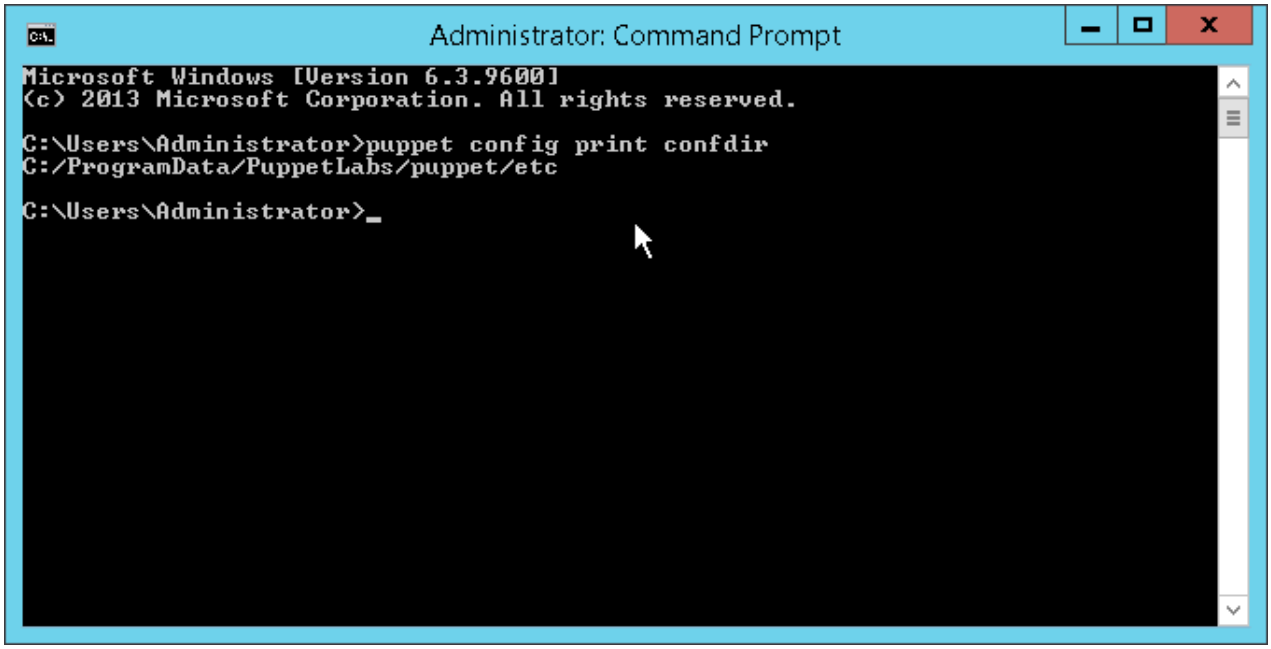
Right-click the **Start** (or apps screen tile) -> **Run as administrator**:



Click **Yes** to allow the command prompt to run with elevated privileges:

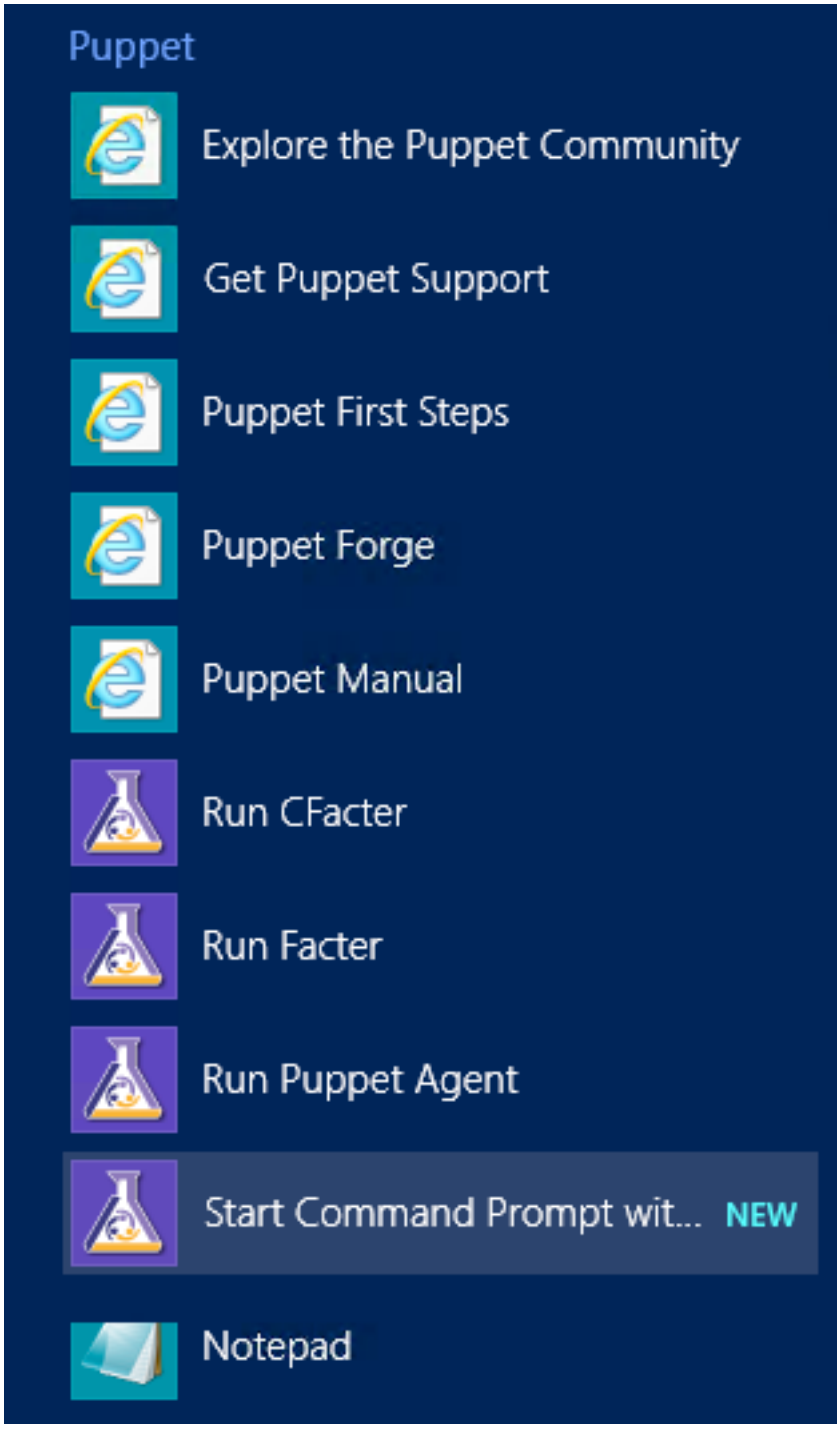


The title bar on the comand prompt window begins with **Administrator**. This means Puppet commands that run from that window can manage the whole system.



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, since 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.

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 will cause 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 once 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 may 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 may not be the most convenient for you.

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.

Since 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 will be 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 PuppetDB's 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 may 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 may not be the most convenient for you.

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 one of the following commands:

Bash:

```
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"
```

PowerShell and the `New-EventLog` cmdlet:

```
if ([System.Diagnostics.Eventlog]::SourceExists("puppet")) { Remove-EventLog
 -Source 'puppet' } & New-EventLog -Source puppet -LogName Puppet
```

Note that for agents older than 5.5.17 on the 5.5.x stream, use the same Bash command listed above, but the following PowerShell command instead:

```
if ([System.Diagnostics.Eventlog]::SourceExists("puppet")) { Remove-
EventLog -Source 'puppet' } & New-EventLog -Source puppet -LogName
 Puppet  -MessageResource "C:\Program Files\Puppet Labs\Puppet\puppet\bin
\puppetres.dll"
```

After you add the registry key, you need to reboot your machine for the logging to be redirected.

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 will not be moved; only new messages will be 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 more information about Puppet's architecture, see Overview of Puppet's architecture — in particular, read the note about differences and trade-offs between agent/master and `puppet apply`.

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 will use 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.

Resource type	Details
`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. 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 will create 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 will be saved, but the default location is `/var/log/messages` on Linux, `/var/log/system.logon` 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 will process 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 once an 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.

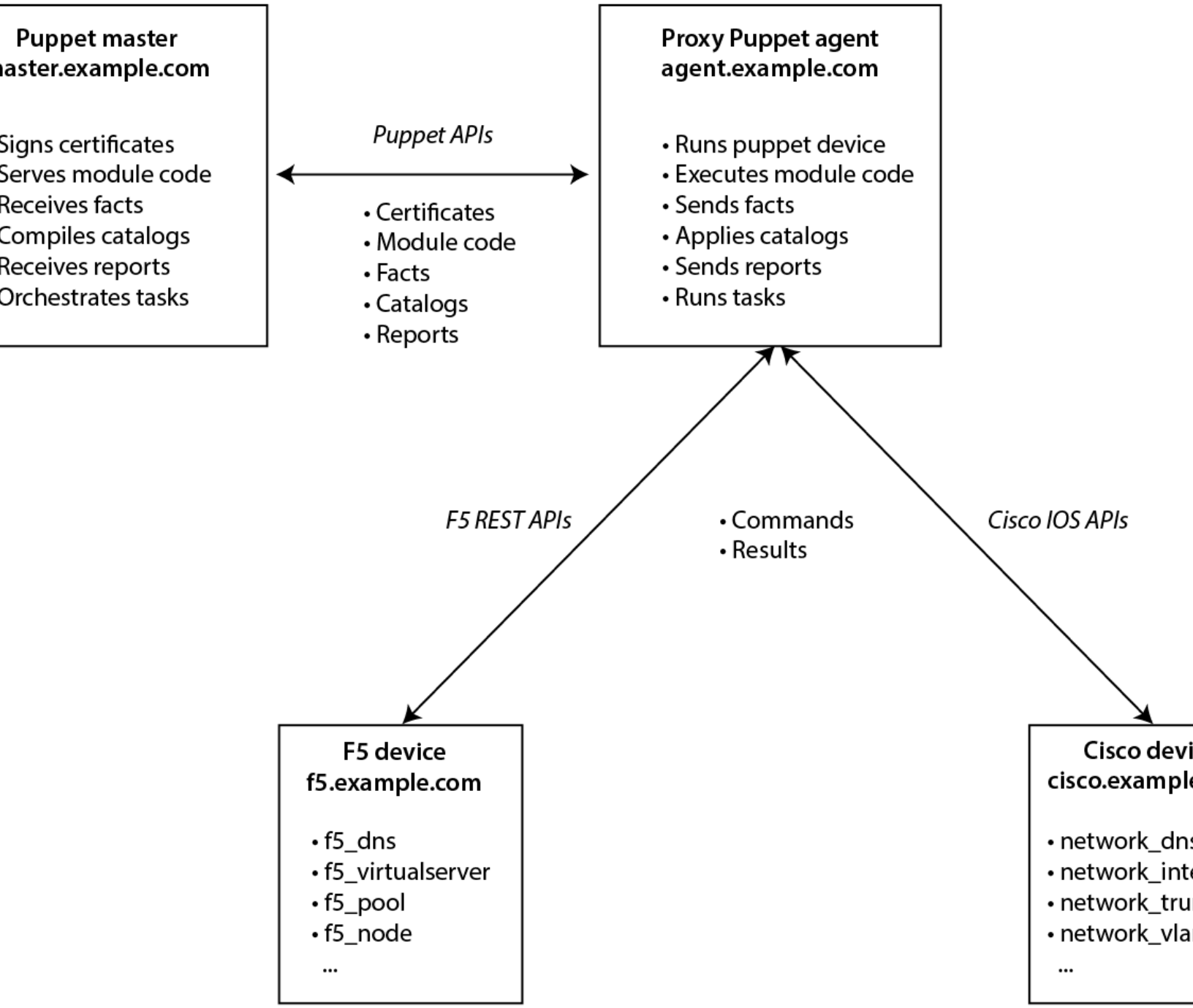
Puppet device runs on both *nix and Windows. The Puppet device application combines some of the functionality of the Puppet apply and Puppet resource applications. For details about running the Puppet device application, see the `puppet device` man page.

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 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 will be 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>` will 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 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

Server-specific Puppet API endpoints

Status API endpoints

Metrics API endpoints

Developer information

The Puppet language

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. You should have one goal, with all your classes and parameters focused on achieving it.

Your module is a piece of software

At least, you should 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 own 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 indention 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.

A string does not have to be in single quotes if it:

- Contains variables.
- Contains single quotes.
- Contains escaped characters not supported by single-quoted strings.
- Is an enumerable set of options, such as present/absent, in which case the single quotes are optional.

Enclose all variables in braces when interpolated in a string.

For example:

Good:

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

Bad:

```
"/etc/$file.conf"
"$facts['operatingsystem'] is not supported by $module_name"
```

When a string contains single quotes, enclose it in double quotes, rather than using escape-character single quotes, unless doing so would require an inconvenient amount of additional escape characters.

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')
```

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.

Related information

Modules on page 115
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/foo':
    content => 'foos content',
}
```

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/foo':
    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 once 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',;

  '/foo':
    user => 'owner',;

  '/bar':
    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,;

  '/foo':
    user => 'owner',;

  '/bar':
    user => 'staff',;
}
```

Good: Spell out 'magic' iteration:

```
['/foo', '/bar'].each |$path| {
  file { $path:
    ensure => 'file',
  }
}
```

Good: Spell out 'magic' iteration:

```
$array_of_paths.each |$path| {
  file { $path:
    ensure => 'file',
  }
}
```

Bad:

```
file {
  '/foo':
    ensure => 'file',
    user   => owner,
    mode   => '0666',;

  '/bar':
    ensure => 'file',
    user   => staff,
    mode   => '0774',;
}

file { ['/foo', '/bar']:
  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 { '/foo':
  ensure => 'file',
  owner  => 'root',
}
```

Bad: Too many levels of indentation:

```
file { '/foo':
```

```
    ensure => 'file',
    owner  => 'root',
}
```

Bad: No indentation:

```
file { '/foo':
ensure => 'file',
owner  => 'root',
}
```

Bad: Improper and non-uniform indentation:

```
file { '/foo':
  ensure => 'file',
   owner => 'root',
}
```

Bad: Indented the wrong direction:

```
  file { '/foo':
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,;

  '/foo':
    ensure => 'file',
    owner  => 'root',
}
```

Defined resource types

Since 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 foo
class foo {
  include bar
}
# end of file
```

Bad: `manifests/init.pp`:

```
class foo {
  #...
}
include bar
```

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. See 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.

You should 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 type declarations, 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.

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, you should 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',
) {

}
```

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',
) {

}
```

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.

Related information

Modules on page 115
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 uppercased 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:

```
$foo_bar
$some_long_variable
$foo_bar123
```

Bad:

```
$fooBar
$someLongVariable
$foo-bar123
```

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 will 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.

Values and data types

Templates

Advanced constructs

Advanced Puppet language constructs help you write simpler and more effective Puppet code by reducing complexity.

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

The general form of a resource collector is:

- 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 will 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 will realize 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 once, 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 only be managed 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 only managed once.

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 will only manage that resource once.

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 will also realize 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 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 once 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.

CAUTION: Each exported resource must be globally unique across every single node. When 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).

Since 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

Writing custom functions

Writing functions (modern Ruby API)

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 once, 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 may well 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.

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 will produce 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

Once 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 will be 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. Since 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 environment 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 will be 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 135
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 147
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 140
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 152

In Hiera you can insert, or interpolate, the value of a variable into a string, using the syntax `%{variable}`.

Hiera hierarchies on page 133
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 156

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 will receive.

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 once as this makes the facts for that node available to the `lookup` command (otherwise you will 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 135
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 133

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"
```

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 will 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 133
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 153

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:

```
{% raw %}
- name: "Domain or network segment"
    glob: "network/**/{%{facts.networking.domain},
%{facts.networking.interfaces.en0.bindings.0.network}}.yaml"
{% endraw %]
```

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 will be 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 152
In Hiera you can insert, or interpolate, the value of a variable into a string, using the syntax `%{variable}`.

The hierarchy on page 136
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 152
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 160
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 152

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 156
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`; disabled by default.
  - `'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 (since 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. Note that this option is disabled by default due to a known issue that causes it to be ineffective in hierarchies more than three levels deep. For more information, see Puppet 5 known issues.
  - `'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 155

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 once.

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 make 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 159
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 once 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 will lookup 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 will be 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 more 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 will win. The rationale for allowing this is that you may be using the same hash of options multiple times, and you may 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) will fail. 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 will use merge behavior from the data sources if present, and will otherwise do 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) will fail.
- `<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 will check this hash for the requested key before giving up. You can combine this with `default_value` or a lambda, which will be 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 will check 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 will pass 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 will use 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. It needs to be run 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 will use 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` 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 'foo' while merging:

```
puppet lookup --node agent.local --merge deep --knock-out-prefix foo
 key_name_one key_name_two
```

To lookup `key_name` with agent.local's facts, and return a default value of `bar` if nothing was found:

```
puppet lookup --node agent.local --default bar key_name
```

To see an explanation of how the value for `key_name` would be 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 will use 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, since 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 once, 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 once
- 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 once, 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 will contain a `path` when the entry in hiera.yaml is using `path`, `paths`, `glob`, `globs`, or `mapped_paths`, and the backend will receive one call per path to an existing file. When the entry in `hiera.yaml` is using `uri` or `uris`, the options hash will have a `uri` key, and the backend function is called once 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 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 may 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 152
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 163
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 data

Access hash and array elements using a key.subkey notation on page 159
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 may 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 will never call 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 data

Configuring a hierarchy level: hiera-eyaml on page 144
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 152

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 will result in `not_found`, and will prevent 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 will then decrypt 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.

Since 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. **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 156
The `lookup` function uses Hiera to retrieve a value for a given key.

Config file syntax on page 140
The `hiera.yaml` file is a YAML file, containing a hash with up to four top-level keys.

Writing new data backends on page 160
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 135
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 168
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 145
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 140
The `hiera.yaml` file is a YAML file, containing a hash with up to four top-level keys.

Custom backends overview on page 160
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 will consult 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 will 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.

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 will 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 135
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 160
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 146
Hiera supports custom backends.

Configuring a hierarchy level: hiera-eyaml on page 144
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 will 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 140

The `hiera.yaml` file is a YAML file, containing a hash with up to four top-level keys.

Custom backends overview on page 160
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 156
The `lookup` function uses Hiera to retrieve a value for a given key.

Merge behaviors on page 147

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.

Once you have a function, tell Hiera to use it by adding it to the module layer `hiera.yaml`. A simple 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 may want to 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 of 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, since 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` will be 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 156
The `lookup` function uses Hiera to retrieve a value for a given key.

Hiera configuration layers on page 135

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.

Resource types

Reports: Tracking Puppet's activity

Extensions for assigning classes to nodes

Misc. references (settings, functions, etc.)

Man pages

Core tools

Occasionally useful

Niche

HTTP API

Puppet v3 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

SSL and certificates

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.

- Configuring external certificate authority on page 181

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.

- Using an external CA with Puppet Server
- External SSL termination with Puppet Server
- Autosigning certificate requests on page 184

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 Puppet master to retrieve a signed certificate once it's available.

- CSR attributes and certificate extensions on page 187

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 all certificates in a Puppet deployment on page 193

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 certificate authority (CA).

Configuring external certificate authority

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 self-signed CA which directly issues SSL certificates.
- Puppet Server functioning as an intermediate CA of a root self-signed 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.

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
}
```

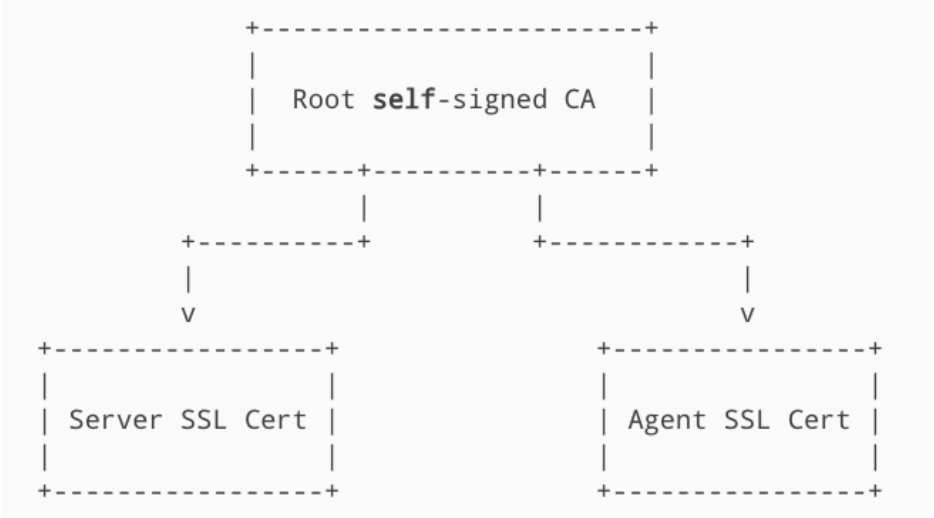
For more information on these settings, seeConfiguring the Web Server Service.

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.

Option 1: 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.



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. You must have your own complete system for issuing and distributing the certificate.

Puppet Server

Configure Puppet Server in three steps:

- Disable the internal CA service.
- Ensure that the certname will 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:

   ```
   # puppetlabs.services.ca.certificate-authority-service/certificate-authority-service
   puppetlabs.services.ca.certificate-authority-disabled-service/certificate-authority-disabled-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 shouldn'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`

Option 2: Puppet Server functioning as an intermediate CA

Puppet Server can operate as an intermediate CA to an external root CA.

For more information, see Using Puppet Server as an intermediate certificate authority.

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 Puppet master to retrieve a signed certificate once it's available.

By default, these CSRs must be manually signed by an admin user, using either the `puppet cert` 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 Puppet master 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 Puppet master's `puppet.conf` file is set to `$confdir/autosign.conf`. This which that the basic autosigning functionality is enabled upon installation.

Depending on your installation method, there might not be a whitelist at that location once the Puppet master 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 can be considered insecure because any host can provide any certname when requesting a certificate. Only use it 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 will run an external policy executable every time it receives a CSR. This executable will examine the CSR and tell 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 will be 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 will pass it the certname of the request (as a command line argument) and the PEM-encoded CSR (on stdin), and will expect a `0` (approved) or non-zero (rejected) exit code.

Once 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 will contain 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 will be as good as your handling of the keys. As long as it's impractical for an attacker to acquire a PSK, it will be 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 will be 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 will run once for each incoming CSR.

	• It will be executed by the Puppet master process and will run as the same user as the Puppet master. • The Puppet master process will block 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. Alternatively, signing requests consume JRubies on a Puppet Server master but might not block all requests while the pool contains available JRubies, and won't block non-Ruby requests.
Arguments	• The executable must allow a single command line argument. This argument will be the Subject CN (certname) of the incoming CSR. • No other command line arguments should be provided. • The Puppet master should never fail to provide this argument.
Stdin	• The executable will receive the entirety of the incoming CSR on its stdin stream. The CSR will be encoded in `pem` format. • The stdin stream will contain nothing but the complete CSR. • 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. • The Puppet master will treat all non-zero exit statuses as equivalent.
Stdout and stderr	• Anything the executable emits on stdout or stderr will be copied to the Puppet master's log output at the `debug` log level. Puppet will otherwise ignore 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 Recovering From Failed Data Embedding below.

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 only embedded 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 `puppet cert 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.

If you use Puppet Server 2.5.0 or higher, you can also sign requests using authorization extensions and the `--allow-authorization-extensions` flag for `puppet cert sign`.

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>]`.

Any OIDs in the ppRegCertExt range appear using their short names. 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 will 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 `puppet cert 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).
- The `puppet cert print` command does display any extensions in a signed certificate, under the `X509v3 extensions` section.

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 will 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 only enables 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
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

AWS 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
```

Assuming your image has the `erb` binary available, 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 once 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.
- Check whether a CSR is present in `$ssldir/certificate_requests/<name>.pem`. If it exists, delete it.
- Check whether a certificate is present in `$ssldir/certs/<name>.pem`. If it exists, delete it.

On the CA Puppet master:

- Check whether a signed certificate exists. Use `puppet cert list --all` to see the complete list. If it exists, revoke and delete it with `puppet cert clean <name>`.
- Check whether a CSR for the node exists in `$ssldir/ca/requests/<name>.pem`. If it exists, delete it.

After you've done that, you can start over.

Regenerating all 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 certificate authority (CA).

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: If you're visiting this page to remediate your Puppet Enterprise deployment due to CVE-2014-0160, also known as Heartbleed, see this announcement for additional information and links to more resources. Before applying these instructions, please note that this is a non-trivial operation that contains some manual steps and will require you to replace certificates on every agent node managed by your Puppet master.

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 will leave 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

Regardless of your situation, regenerating your certs involves the following three steps, described in detail in the sections below:

1. On your master, you'll clear the certs and security credentials, regenerate the CA, and then regenerate the certs and security credentials.
2. You'll clear and regenerate certs and security credentials for any extensions.
3. You'll clear and regenerate certs and security credentials for all agent nodes.

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 just need to replace a few agent certificates, use the `puppet cert clean` command on your Puppet master and then follow step 3 for any agents that need to be replaced.

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. Generate the Puppet master's new certs:

```
sudo puppet master --no-daemonize --verbose
```

7. When you see the message `Notice: Starting Puppet master <VERSION>`, type **CTRL** + **C**.
8. Start the Puppet master service by running:

```
sudo puppet resource service puppetserver ensure=running
```

9. 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.

MCollective

MCollective often uses SSL certificates from Puppet's CA. If you are replacing your Puppet CA and are using the same certs for MCollective, refer to the standard deployment guide and re-do any steps involving security credentials. You'll generally need to replace client certificates, your server keypair, and the ActiveMQ server's keystore and truststore.

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 PuppetDB's Puppet Query Language.

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.

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

The Puppet agent and master communicate via host-verified HTTPS.

Note: Before the agent has a certificate, it makes unverified requests. During these requests, the agent does not identify itself to the master, and does not check the master's certificate against the Certificate Authority (CA). After the agent gets a certificate, requests are host-verified unless stated otherwise.

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 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 only verified HTTPS connections, so it does not cache the unverified connections a new agent makes to request a new certificate. It also does not cache connections when a custom HTTP connection class has been specified.

When an HTTP server disables persistent connections, Puppet requests that the connection is kept open, but the server declines by sending `Connection: close` in the HTTP response. Puppet 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:
   - If the agent doesn't have a private key at `$ssldir/private_keys/<NAME>.pem`, one is generated.
   - If the agent doesn't have a copy of the CA certificate at `$ssldir/certs/ca.pem`, it is fetched using an unverified GET request to `/certificate/ca`.

     Note: This could be vulnerable to Man-In-The-Middle attacks. You can make this step unnecessary by distributing the CA cert as part of your server provisioning process, so that agents never ask for a CA cert over the network. If you do this, an attacker could temporarily deny Puppet service to brand new nodes, but would be unable to take control of them with a rogue Puppet master.
   - If the agent has a signed certificate at `$ssldir/certs/<NAME>.pem`, skip to Request node object and switch environments. If the agent has a cert but it doesn't match the private key, an error is generated.
2. Get a certificate, if needed. If the agent has submitted a certificate signing request (CSR), and if autosign is not enabled, an admin user will need to run `puppet cert sign <NAME>` on the CA Puppet master before the agent can fetch a signed certificate. Because incoming CSRs are unverified, use fingerprints to prove them, by comparing `puppet agent --fingerprint` on the agent to `puppet cert list` on the CA master.
   - Try to fetch a signed certificate from the master using an unverified GET request to `/puppet-ca/v1/certificate/<NAME>` . If the request is successful, skip the rest of this section and continue to Request node object and switch environments. If the cert doesn't match the private key, exit with an error.
   - Check if there's already a CSR for the agent.
     - Check in `$ssldir/certificate_requests/<NAME>.pem`.
     - Check with the master whether the agent has already requested a certificate signing. The agent might have lost the local copy of the request. Make an unverified GET request to `/puppet-ca/v1/certificate_request/<NAME>`.
     - If the CSR exists, the agent exits the process because user intervention is needed on the master. If `waitforcert` is enabled, the agent waits a few seconds and returns to the first step of this section to request a cert.
   - If no cert request was made, return to the first step of this section to request a cert.
3. 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.

4. 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.
5. 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.
6. 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, will amplify either issue.
7. 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, since each node only receives 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 its own catalog and then applies it, so it plays the role of both master and agent.

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.
   - Once the master has 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.
   - Less commonly, the `ldap` node terminus fetches information from an LDAP database.
   - 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 node's catalog, and any classes declared in the code are loaded and declared.

     Note: Classes are usually 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 top-scope variables.) Resources in the code 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.

For more information, see Configuration, Indirection, Writing external node classifiers, The LDAP node classifier, Language: node definitions, Language: Scope, Directories: The modulepath (default config).

Related information

Exported resources on page 127

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