



Open source Puppet 7.5.0

Contents

Puppet 7.5.0.....	6
Introduction to Puppet.....	6
What is Puppet?.....	6
Why use Puppet desired state management?.....	7
Key concepts behind Puppet.....	8
The Puppet platform.....	8
Open source Puppet vs Puppet Enterprise (PE).....	11
The Puppet ecosystem.....	11
Use cases.....	12
Puppet platform lifecycle.....	13
Navigating the documentation.....	15
Archived Puppet documentation.....	16
Glossary.....	16
 Release notes.....	 16
Puppet release notes.....	17
Puppet known issues.....	28
Facter release notes.....	29
Facter known issues.....	34
Puppet Server release notes.....	34
Puppet Server 7.1.0.....	34
Puppet Server 7.0.3.....	35
Puppet Server 7.0.2.....	35
Puppet Server 7.0.0.....	35
Puppet Server known Issues.....	36
Cipher updates in Puppet Server 6.5.....	36
Server-side Ruby gems might need to be updated for upgrading from JRuby 1.7.....	36
Potential JAVA ARGS settings.....	36
tmp directory mounted noexec.....	37
Puppet Server Fails to Connect to Load-Balanced Servers with Different SSL Certificates.....	37
What's new since Puppet 6?.....	37
 Installing and configuring.....	 39
Installing and upgrading.....	39
System requirements.....	39
Installing Puppet.....	41
Installing and configuring agents.....	43
Manually verify packages.....	50
Managing Platform versions.....	53
Upgrading.....	53
Configuring Puppet settings.....	56
Puppet settings.....	56
Key configuration settings.....	59
Puppet's configuration files.....	62
Configuring Puppet Server.....	72
Adding file server mount points.....	88
Checking the values of settings.....	90

Editing settings on the command line.....	92
Differing behavior in puppet.conf.....	93

Components and functionality..... 96

Puppet Server.....	97
About Puppet Server.....	97
Puppet Server release notes.....	100
Deprecated features.....	102
Compatibility with Puppet agent.....	107
Installing Puppet Server.....	107
Configuring Puppet Server.....	108
Differing behavior in puppet.conf.....	131
Using and extending Puppet Server.....	134
Known issues and workarounds.....	163
Developer information.....	164
Puppet Server HTTP API.....	172
Metrics API endpoints.....	185
Status API endpoints.....	189
Certificate Clean.....	192
Server-specific Puppet API endpoints.....	192
Administrative API endpoints.....	202
Certificate authority and SSL.....	203
PuppetDB.....	223
Factor.....	223
Custom facts overview.....	224
Writing custom facts.....	230
External facts.....	235
Configuring Factor with factor.conf.....	238
Hiera.....	241
About Hiera.....	241
Getting started with Hiera.....	245
Configuring Hiera.....	249
Creating and editing data.....	256
Looking up data with Hiera.....	264
Writing new data backends.....	269
Upgrading to Hiera 5.....	276
Environments.....	289
About environments.....	289
Creating environments.....	291
Environment isolation.....	294
Important directories and files.....	296
Code and data directory (codedir).....	296
Config directory (confdir).....	297
Main manifest directory.....	298
The modulepath.....	299
SSL directory (ssldir).....	301
Cache directory (vardir).....	303
Puppet services and tools.....	305
Puppet commands.....	305
Running Puppet commands on Windows.....	307
primary Puppet server.....	311
Puppet agent on *nix systems.....	314
Puppet agent on Windows.....	317
Puppet apply.....	320
Puppet device.....	322

Classifying nodes.....	329
Puppet reports.....	332
Reporting.....	332
Report reference.....	333
Writing custom report processors.....	334
Report format.....	335
Puppet's internals.....	341
Agent-server HTTPS communications.....	341
Catalog compilation.....	342

Developing Puppet code..... 345

The Puppet language.....	345
Puppet language overview.....	348
Puppet language syntax examples.....	350
The Puppet language style guide.....	355
Files and paths on Windows.....	379
Code comments.....	380
Variables.....	380
Resources.....	383
Resource types.....	391
Relationships and ordering.....	405
Classes.....	411
Defined resource types.....	418
Bolt tasks.....	421
Expressions and operators.....	422
Conditional statements and expressions.....	431
Function calls.....	438
Node definitions.....	441
Facts and built-in variables.....	443
Reserved words and acceptable names.....	449
Custom resources.....	454
Custom functions.....	483
Values, data types, and aliases.....	504
Templates.....	553
Advanced constructs.....	569
Details of complex behaviors.....	583
Modules.....	592
Modules overview.....	592
Plug-ins in modules.....	596
Module cheat sheet.....	598
Installing and managing modules from the command line.....	599
Beginner's guide to writing modules.....	607
Module metadata.....	612
Documenting modules.....	618
Documenting modules with Puppet Strings.....	622
Puppet Strings style guide.....	629
Publishing modules.....	635
Contributing to Puppet modules.....	638
Designing system configs: roles and profiles.....	641
The roles and profiles method.....	641
Roles and profiles example.....	645
Designing advanced profiles.....	648
Designing convenient roles.....	664
Puppet Forge.....	668
Puppet Development Kit (PDK).....	668

Puppet VSCode extension.....	668
Orchestration in Puppet.....	668
Example configurations.....	669
Manage NTP.....	669
Manage sudo.....	672
Manage DNS.....	675
Manage firewall rules.....	678
Forge examples.....	681
References.....	682
Experimental features.....	682
Msgpack support.....	683
Metaparameter reference.....	684

Welcome to Puppet 7.5.0

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 links
Getting started Introduction to Puppet Release notes Glossary	Docs for related Puppet products Bolt Puppet Enterprise Continuous Delivery for Puppet Enterprise Puppet Development Kit Puppet VSCode extension
Install and configure Puppet Install Puppet Configure Puppet settings	Share and contribute Puppet Forge Puppet Community Open source projects on GitHub
Learn about the Puppet platform Puppet Server PuppetDB Facter	Learn more about Puppet Blog posts about Puppet Puppet training
Develop Puppet code Puppet language Modules overview	

Introduction to Puppet

Welcome to the open source *Puppet* documentation!

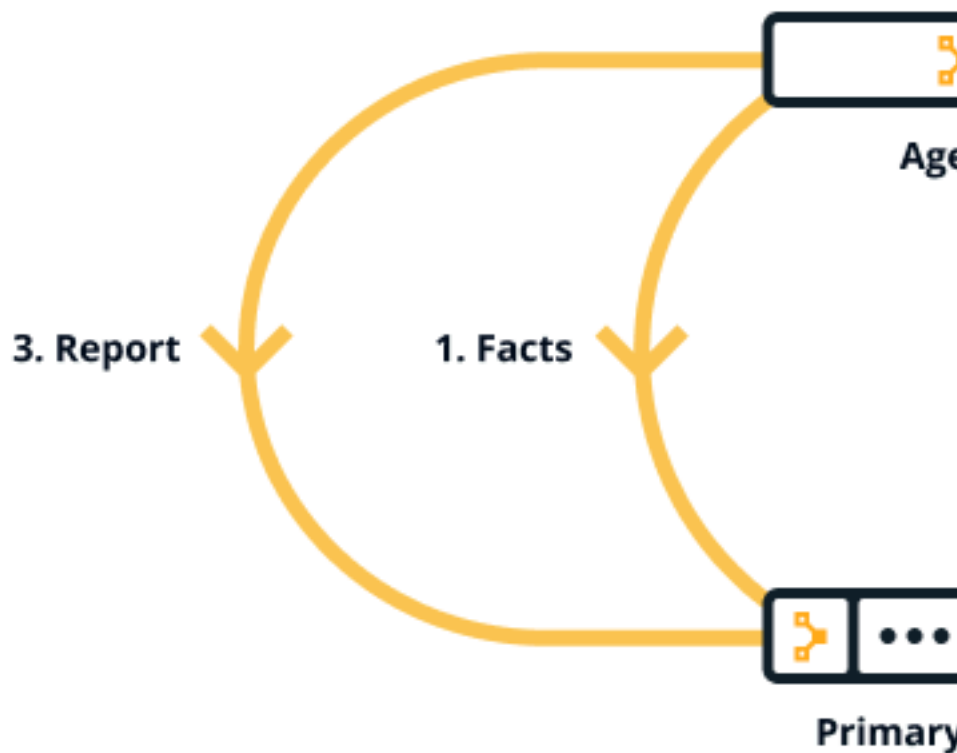
This introduction is intended for new users to Puppet. We go over what Puppet is, what problems it solves, and the concepts and practices that are key to being successful with Puppet.

What is Puppet?

Puppet is a tool that helps you manage and automate the configuration of servers.

When you use Puppet, you define the *desired state* of the systems in your infrastructure that you want to manage. You do this by writing infrastructure code in Puppet's Domain-Specific Language (DSL) — Puppet Code — which you can use with a wide array of devices and operating systems. Puppet code is *declarative*, which means that you describe the desired state of your systems, not the steps needed to get there. Puppet then *automates* the process of getting these systems into that state and keeping them there. Puppet does this through Puppet *primary server* and a Puppet *agent*. The Puppet primary server is the server that stores the code that defines your desired state. The Puppet agent translates your code into commands and then executes it on the systems you specify, in what is called a Puppet run.

The diagram below shows how the server-agent architecture of a Puppet run works.



The primary server and the agent are part of the Puppet platform, which is described in [The Puppet platform](#) — along with facts, catalogs and reports.

Why use Puppet desired state management?

There are many benefits to implementing a declarative configuration tool like Puppet into your environment — most notably *consistency* and *automation*.

- **Consistency.** Troubleshooting problems with servers is a time-consuming and manually intensive process. Without configuration management, you are unable to make assumptions about your infrastructure — such as which version of Apache you have or whether your colleague configured the machine to follow all the manual steps correctly. But when you use configuration management, you are able to validate that Puppet applied the desired state you wanted. You can then assume that state has been applied, helping you to identify why your model failed and what was incomplete, and saving you valuable time in the process. Most importantly, once you figure it out, you can add the missing part to your model and ensure that you never have to deal with that same problem again.
- **Automation.** When you manage a set of servers in your infrastructure, you want to keep them in a certain state. If you only have to manage homogeneous 10 servers, you can do so with a script or by manually going into each server. In this case, a tool like Puppet may not provide much extra value. But if you have 100 or 1,000 servers, a mixed environment, or you have plans to scale your infrastructure in the future, it is difficult to do this manually. This is where Puppet can help you — to save you time and money, to scale effectively, and to do so securely.

Check out the following video of how a DevOps engineer uses Puppet:

Key concepts behind Puppet

Using Puppet is not just about the tool, but also about a different culture and a way of working. The following concepts and practices are key to using and being successful with Puppet.

Infrastructure-as-code

Puppet is built on the concept of *infrastructure-as-code*, which is the practice of treating infrastructure as if it were code. This concept is the foundation of DevOps — the practice of combining software development and operations. Treating infrastructure as code means that system administrators adopt practices that are traditionally associated with software developers, such as version control, peer review, automated testing, and continuous delivery. These practices that test code are effectively testing your infrastructure. When you get further along in your automation journey, you can choose to write your own unit and acceptance tests — these validate that your code, your infrastructure changes, do as you expect. To learn more about infrastructure-as-code and how it applies to Puppet, see our blog [What is infrastructure as code?](#).

Idempotency

A key feature of Puppet is *idempotency* — the ability to repeatedly apply code to guarantee a desired state on a system, with the assurance that you will get the same result every time. Idempotency is what allows Puppet to run continuously. It ensures that the state of the infrastructure always matches the desired state. If a system state changes from what you describe, Puppet will bring it back to where it is meant to be. It also means that if you make a change to your desired state, your entire infrastructure automatically updates to match. To learn more about idempotency, see our [Understanding idempotency](#) documentation.

Agile methodology

When adopting a tool like Puppet, you will be more successful with an *agile methodology* in mind — working in incremental units of work and reusing code. Trying to do too much at once is a common pitfall. The more familiar you get with Puppet, the more you can scale, and the more you get used to agile methodology, the more you can democratize work. When you share a common methodology, a common pipeline, and a common language (the Puppet language) with your colleagues, your organization becomes more efficient at getting changes deployed quickly and safely.

Git and version control

Git is a *version control* system that tracks changes in code. While version control is not required to use Puppet, it is highly recommended that you store your Puppet code in a Git repository. Git is the industry standard for version control, and using it will help your team gain the benefits of the DevOps and agile methodologies.

When you develop and store your Puppet code in a Git repository, you will likely have multiple branches — feature branches for developing and testing code and a production branch for releasing code. You test all of your code on a feature branch before you merge it to the production branch. This process, known as Git flow, allows you to test, track, and share code, making it easier to collaborate with colleagues. For example, if someone on your team wants to make a change to an application's firewall requirements, they can create a pull request that shows their proposed changes to the existing code, which everyone on your team can review before it gets pushed to production. This process leaves far less room for errors that could cause an outage. For more information on version control, see the GitHub guides [Git flow](#) and [What is version control?](#).

The Puppet platform

Puppet is made up of several packages. Together these are called the Puppet platform, which is what you use to manage, store and run your Puppet code. These packages include `puppetserver`, `puppetdb`, and `puppet-agent` — which includes *Facter* and *Hiera*.

Puppet is configured in an agent-server architecture, in which a primary node (system) controls configuration information for one or more managed agent nodes. Servers and agents communicate by HTTPS using SSL certificates. Puppet includes a built-in certificate authority for managing certificates. Puppet Server performs the role of the primary node and also runs an agent to configure itself.

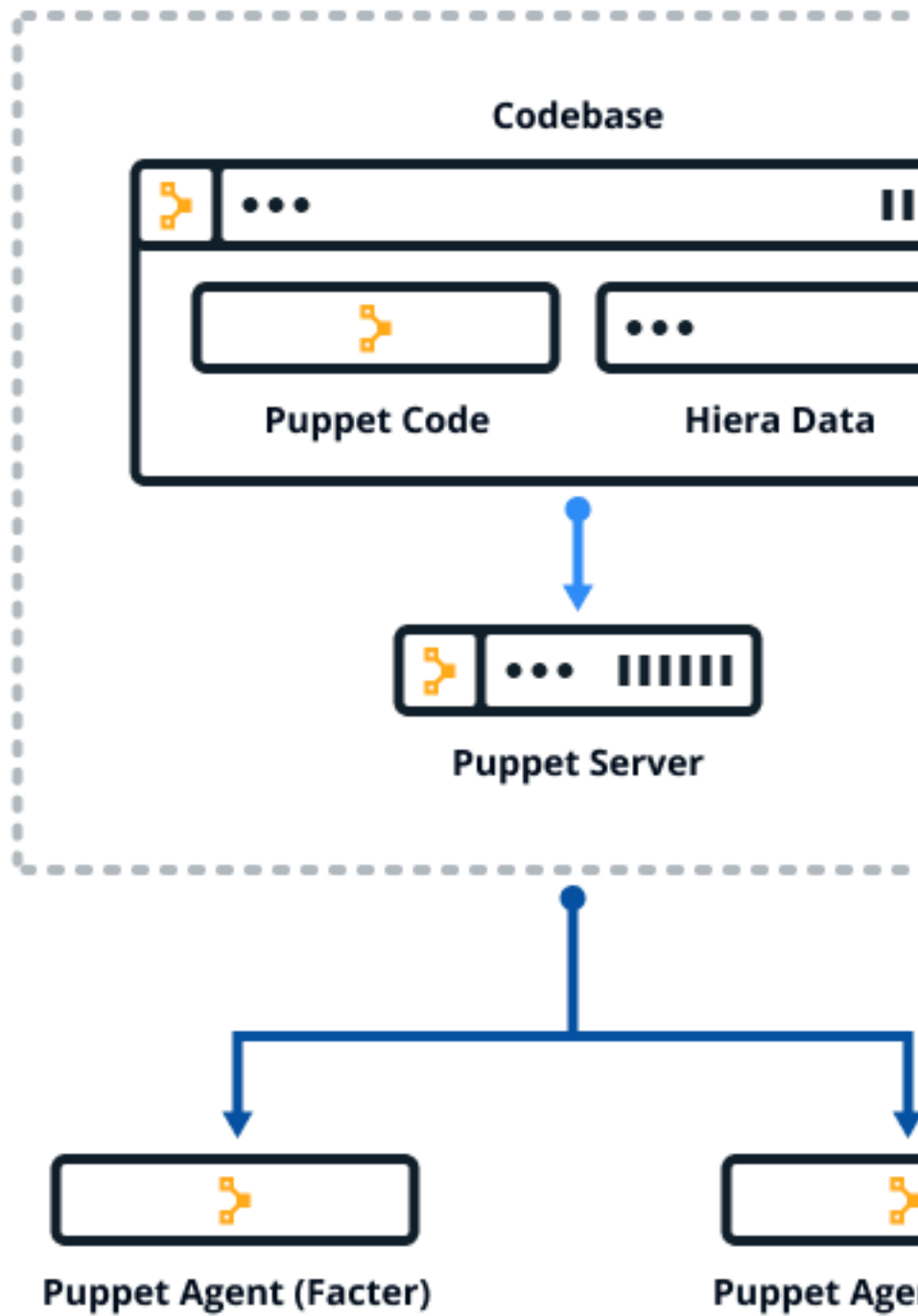
Facter, Puppet's inventory tool, gathers *facts* about an agent node such as its hostname, IP address, and operating system. The agent sends these facts to the primary server in the form of a special Puppet code file called a *manifest*. This is the information the primary server uses to compile a *catalog* — a JSON document describing the desired state of a specific agent node. Each agent requests and receives its own individual catalog and then enforces that desired state on the node it's running on. In this way, Puppet applies changes all across your infrastructure, ensuring that each node matches the state you defined with your Puppet code. The agent sends a report back to the primary server.

You keep nearly all of your Puppet code, such as manifests, in *modules*. Each module manages a specific task in your infrastructure, such as installing and configuring a piece of software. Modules contain both code and data. The data is what allows you to customize your configuration. Using a tool called *Hiera*, you can separate the data from the code and place it in a centralized location. This allows you to specify guardrails and define known parameters and variations, so that your code is fully testable and you can validate all the edge cases of your parameters. If you have just joined an existing team that uses Puppet, take a look at how they organize their Hiera data.

All of the data generated by Puppet (for example facts, catalogs, reports) is stored in the *Puppet database* (PuppetDB). Storing data in PuppetDB allows Puppet to work faster and provides an API for other applications to access Puppet's collected data. Once PuppetDB is full of your data, it becomes a great tool for infrastructure discovery, compliance reporting, vulnerability assessment, and more. You perform all of these tasks with PuppetDB queries.

The diagram below shows how the Puppet package components fit together.

Primary Server



Related information

[Facter](#) on page 223

Facter is Puppet's cross-platform system profiling library. It discovers and reports per-node facts, which are available in your Puppet manifests as variables.

[Hier](#) on page 241

Hiera is a built-in key-value configuration data lookup system, used for separating data from Puppet code.

[Certificate authority and SSL](#) on page 203

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.

Open source Puppet vs Puppet Enterprise (PE)

Puppet Enterprise (PE) is the commercial version of Puppet and is built on top of the open source Puppet platform. Both products allow you to manage the configuration of thousands of nodes. Open source Puppet does this with desired state management. PE provides an imperative, as well as declarative, approach to infrastructure automation.

If you have a complex or large infrastructure that is used and managed by multiple teams, PE is a more suitable option, as it provides a graphical user interface, point-and-click code deployment strategies, continuous testing and integration, and the ability to predict the impact of code changes before deployment.

For more information on the differences between open source Puppet and PE, see our [comparison page](#). For additional information on PE, see the [PE documentation](#).

The Puppet ecosystem

Alongside Puppet the configuration tool, there are additional Puppet tools and resources to help you use and be successful. These make up the Puppet ecosystem

Install existing modules from Puppet Forge

Modules manage a specific technology in your infrastructure and serve as the basic building blocks of Puppet desired state management. On the Puppet Forge, there is a module to manage almost any part of your infrastructure. Whether you want to manage packages or patch operating systems, a module is already set up for you. See each module's README for installation instructions, usage, and code examples.

When using an existing module from the Forge, most of the Puppet code is written for you. You just need to install the module and its dependencies and write a small amount of code (known as a profile) to tie things together. Take a look at our [Getting started with PE guide](#) to see an example of writing a profile for an existing module. For more information about existing modules, see the [module fundamentals documentation](#) and [Puppet Forge](#).

Develop existing or new modules with Puppet Development Kit (PDK)

You can write your own Puppet code and modules using Puppet Development Kit (PDK), which is a framework to successfully build, test and validate your modules. Note that most Puppet users won't have to write full Puppet code at all, though you can if you want to. For installation instructions and more information, see the [PDK documentation](#).

Write Puppet code with the VSCode extension

The *Puppet VSCode extension* makes writing and managing Puppet code easier and ensures your code is high quality. Its features include Puppet DSL intellisense, linting, and built-in commands. You can use the extension with Windows, Linux, or macOS. For installation instructions and a full list of features, see the [Puppet VSCode extension documentation](#).

Run acceptance tests with Litmus

Litmus is a command line tool that allows you to run acceptance tests against Puppet modules for a variety of operating systems and deployment scenarios. Acceptance tests validate that your code does what you intend it to do. For more information, see the [Litmus documentation](#).

Use cases

Puppet Forge has existing modules and code examples that assist with automating the following use cases:

- **Base system configuration**
 - Including [registry](#), [NTP](#), [firewalls](#), [services](#)
- **Manage web servers**
 - Including [apache](#), [tomcat](#), [IIS](#), [nginx](#)
- **Manage database systems**
 - Including [Oracle](#), [Microsoft SQL Server](#), [MySQL](#), [PostgreSQL](#)
- **Manage middleware/application systems**
 - Including [Java](#), [WebLogic/Fusion](#), [IBM MQ](#), [IBM IIB](#), [RabbitMQ](#), [ActiveMQ](#), [Redis](#), [ElasticSearch](#)
- **Source control**
 - Including [Github](#), [Gitlab](#)
- **Monitoring**
 - Including [Splunk](#), [Nagios](#), [Zabbix](#), [Sensu](#), [Prometheus](#), [NewRelic](#), [Icinga](#), [SNMP](#)
- **Patch management**
 - [OS patching](#) on Enterprise Linux, Debian, SLES, Ubuntu, Windows
- **Package management**
 - Linux: Puppet integrates directly with native package managers
 - Windows: Use Puppet to install software directly on Windows, or integrate with [Chocolatey](#)
- **Containers and cloud native**
 - Including [Docker](#), [Kubernetes](#), [Terraform](#), [OpenShift](#)
- **Networking**
 - Including [Cisco Catalyst](#), [Cisco Nexus](#), [F5](#), [Palo Alto](#), [Barracuda](#)
- **Secrets management**
 - Including [Hashicorp Vault](#), [CyberArk Conjur](#), [Azure Key Vault](#), [Consul Data](#)

See each module's README for installation, usage, and code examples.

If you don't see your use case listed above, have a look at the following list to see what else we might be able to help you with:

- **Continuous integration and delivery of Puppet code**
 - *Continuous Delivery for Puppet Enterprise (PE)* offers a prescriptive workflow to test and deploy Puppet code across environments. To harness the full power of PE, you need a robust system for testing and deploying your Puppet code. Continuous Delivery for PE offers prescriptive, customizable work flows and intuitive tools for Puppet code testing, deployment, and impact analysis — so you know how code changes will affect your infrastructure before you deploy them — helping you ship changes and additions with speed and confidence. For more information, see [CD4PE](#).
- **Incident remediation**
 - If you need to minimize the risk of external attacks and data breaches by increasing your visibility into the vulnerabilities across your infrastructure, take a look at *Puppet Remediate*. With Remediate, you can eliminate the repetitive and error-prone steps of manual data handovers between teams. For more information, see [Puppet Remediate](#).
- **Integrate Puppet into your existing workflows**
 - Take a look at our integrations with other technology, including [Splunk](#) and [VMware vRA](#).

Puppet platform lifecycle

The open source Puppet platform is made up of several packages: `puppet-agent`, `puppetserver`, and, optionally, `puppetdb`. Understanding what versions are maintained and which versions go together is important when upgrading and troubleshooting.

Puppet releases and lifecycle

Open source Puppet has two release tracks:

- **Update track:** Puppet versions that are not associated with any PE version get updated minor (or "y") releases about once a month. Releases in this track include fixes and new features, but typically do not get patch (or "z") releases. Each update in this track supersedes the previous minor release. Documentation for the current release is available at puppet.com/docs/puppet/latest. The latest [Release notes](#) on page 16 contain a history of all updates to this release track.
- **Long-term releases:** Puppet versions associated with Puppet Enterprise LTS (long-term support) releases get patch (or "z") releases about quarterly. Each release contains bug and security fixes from several developmental releases, but does not get new features. Versioned documentation for long-term releases is available at puppet.com/docs/puppet/<X.Y> (for example, puppet.com/docs/puppet/6.4).

Important: To ensure that you have the most recent features, fixes, and security patches, update your Puppet version whenever there is a new version in your release track.

The following table lists the maintained Puppet, Puppet Server, and PuppetDB versions, with links to their respective documentation. Developmental releases ('latest') are superseded by new versions about once a month. Open source releases that are associated with PE versions have projected End of Life (EOL) dates.

Puppet version	Puppet Server version	PuppetDB version	Associated PE version	Projected EOL date
7.5.0 (latest)	7.1.0	7.2.0	2021.0.x	Superseded by next developmental release.
6.21.1	6.15.1	6.15.0	2019.8.x	December 2022

Note: To access docs for unmaintained Puppet versions, visit our [Archived Puppet documentation](#) on page 16 page.

For information about Puppet's operating system support, see the [platform support lifecycle](#) page.

Puppet platform packages

The Puppet platform bundles the components needed for a successful deployment. We distribute open source Puppet in the following packages:

Package	Contents
<code>puppet-agent</code>	This package contains Puppet's main code and all of the dependencies needed to run it, including Factor, Hiera, the PXP agent, root certificates, and bundled versions of Ruby and OpenSSL. After it's installed, you have everything you need to run the Puppet agent service and the <code>puppet apply</code> command.

Package	Contents
puppetserver	Puppet Server is a JVM-based application that, among other things, runs instances of the primary Puppet server application and serves catalogs to nodes running the agent service. It has its own version number and might be compatible with more than one Puppet version. This package depends on puppet-agent. After it's installed, Puppet Server can serve catalogs to nodes running the agent service.
puppetdb	PuppetDB (optional) collects data generated by Puppet. It enables additional features such as exported resources , advanced queries, and reports about your infrastructure.
puppetdb-termini	Plugins to connect your primary server to PuppetDB

The puppetserver component of the Puppet platform is available only for Linux. The puppet-agent component is available independently for over 30 platforms and architectures, including Windows and macOS.

Note: Mcollective was removed in Puppet 6.0. If you use Puppet Enterprise, consider [Puppet orchestrator](#). If you use open source Puppet, migrate MCollective agents and filters using tools such as [Bolt](#) and PuppetDB's [Puppet Query Language](#).

puppet-agent component version numbers

Each puppet-agent package contains several components. This table shows the components shipped in this release track, and contains links to available component release notes. Agent release notes are included on the same page as Puppet release notes.

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

puppet-agent	Puppet	Facter	Hiera	Resource API	Ruby	OpenSSL
7.5.0	7.5.0	4.0.52	3.6.0	1.8.13	2.7	1.1.1i
7.4.1	7.4.1	4.0.51	3.6.0	1.8.13	2.7	1.1.1i
7.4.0	7.4.0	4.0.50	3.6.0	1.8.13	2.7	1.1.1i
7.3.0	7.3.0	4.0.49	3.6.0	1.8.13	2.7	1.1.1i
7.1.0	7.1.0	4.0.47	3.6.0	1.8.13	2.7	1.1.1i
7.0.0	7.0.0	4.0.46	3.6.0	1.8.13	2.7	1.1.1g

Primary server and agent compatibility

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

	Server		
Agent	PE 2017.3 through 2018.1 Puppet 5.x	PE 2019.1 through 2019.8 Puppet 6.x	PE 2021.0 and later Puppet 7.x
5.x	#	#	
6.x	#	#	#
7.x	#	#	#

Note: Puppet 5.x has reached end of life and is not actively developed or tested. We retain agent 5.x compatibility with later versions of the server only to enable upgrades.

Navigating the documentation

Puppet maintains a large amount of documentation and learning resources to help you learn Puppet. When navigating the documentation, take note of the following.

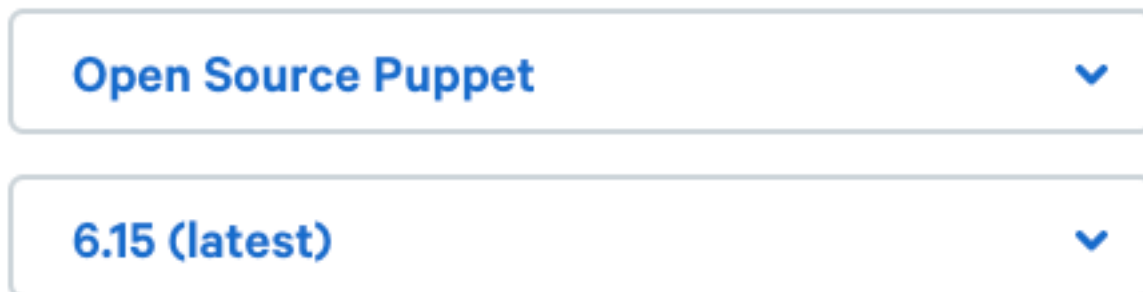
The search bar

The Puppet documentation search bar at the top of the page can be useful when you know exactly what you are looking for. Unlike search engines, it has the added benefit of being able to filter by Puppet product and version. The filter appears on the right side of the page after you search for something.



The version switcher

If you land on a documentation page from an external search engine, make sure you are looking at the correct Puppet version. You can see what version of the documentation you are viewing by looking at the version switcher in the top left corner.



Note that, other than the latest version, we only maintain and update the open source Puppet versions that are built into Puppet Enterprise (PE) versions receiving long-term support. For information on which versions we currently support, see [Puppet packages and versions](#). If we no longer update a page, the following banner is shown across the top.

This version is out of date. For current versions, s

Code examples

We have code examples throughout the docs. These are often general examples designed to help a wide audience. To see more real-life and specific examples, take a look at the relevant module on Puppet Forge. For example, to see what a code example for managing NTP services looks like, take a look at the [NTP README](#). However, if you spot a place in the documentation where you would like more examples, please let us know.

The glossary

If you come across a term that you are unfamiliar with, you will likely find a definition for it in our [glossary](#).

We welcome your feedback!

If you don't see what you need, if something isn't clear enough, or if you spot a mistake, please let the Puppet documentation team know, either by using the star rating at the bottom of the page or by [opening a ticket](#) (you'll need a Jira account). Whichever feedback method you choose, a member of the docs team reads every piece of feedback, so the more specific you can be about the request or issue, the more quickly and easily we'll be able to help and update the documentation. As noted before, we can only update [maintained Puppet versions](#). We greatly appreciate your feedback!

Archived Puppet documentation

Open source Puppet docs for recent end-of-life (EOL) product versions are archived in place, meaning that we continue to host them at their original URLs, but we limit their visibility on the main docs site and no longer update them. You can access archived-in-place docs using their original URLs, or from the links here.

Open Source Puppet docs for EOL versions earlier than those listed here are archived in our open source Puppet [docs archive](#).

Puppet Version	URL
6.4	https://puppet.com/docs/puppet/6.4/puppet_index.html
6.0	https://puppet.com/docs/puppet/6.0/puppet_index.html
4.10	https://puppet.com/docs/puppet/4.10/index.html
3.8	https://puppet.com/docs/puppet/3.8/index.html

Glossary

Definitions of terms used in Puppet documentation

Release notes

These release notes contain important information about the Puppet® 7.5 platform, including Puppet agent, Puppet Server, Facter and PuppetDB.

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

Version numbers use the format X.Y.Z, where:

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

- Z can increase for bug fixes.

The following table lists the maintained Puppet, Puppet Server, and PuppetDB versions. Developmental releases ('latest') are superseded by new versions about once a month. Open source releases that are associated with PE versions have projected End of Life (EOL) dates.

Puppet version	Puppet Server version	PuppetDB version	Associated PE version	Projected EOL date
7.5.0 (latest)	7.1.0	7.2.0	2021.0.x	Superseded by next developmental release.
6.21.1	6.15.1	6.15.0	2019.8.x	December 2022

Note: Security and vulnerability announcements are posted at <https://puppet.com/docs/security-vulnerability-announcements>.

See the following pages for a full list of platform release notes in the Puppet 7 series.

- [Puppet release notes](#) on page 17

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

- [Puppet known issues](#) on page 28

These are the known issues in this version of Puppet.

- [Facter release notes](#) on page 29

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

- [Facter known issues](#) on page 34

These are the known issues in this version of Facter.

- [Puppet Server release notes](#) on page 34

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

- [Puppet Server known Issues](#) on page 36

These are the known issues in this version of Puppet Server.

- [PuppetDB release notes \(link\)](#)

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

- [What's new since Puppet 6?](#) on page 37

These are the major new features, enhancements, deprecations, and removals since the Puppet 6 release.

Puppet release notes

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

Puppet 7.5.0

Released 16 March 2021.

We would like to thank the following Puppet community members for their contributions to this release: [smokris](#) and [priv-kweihmann](#).

New features

The puppet ssl show command

The puppet ssl show command prints the full-text version of a host's certificate, including extensions. [PUP-10888](#)

The `ciphers` setting

The `ciphers` setting configures which TLS ciphersuites the agent supports. The default set of ciphersuites is the same, but you can now make the list of ciphersuites more restricted, for example, to only accept TLS v1.2 or greater ciphersuites. [PUP-10889](#)

The `GlobalSignRoot CA R3`

This release adds the `GlobalSignRoot CA R3` certificate for `rubygems.org`. [PA-3525](#)

Resolved issues

The splat operator in a virtual query is not supported

This release fixes a regression in Puppet 7.x that prevented the splat operator from being used to override resource attributes in a resource collector. [PUP-10951](#)

Windows package provider continues to read `DisplayVersion` key after it is embedded `NULL`

Previously, Puppet would not stop reading the registry at the correct `WCHAR_NULL` because it was encoded to UTF-16LE, causing Puppet to read bad data and fail. This is now fixed. [PUP-10943](#)

Listing environments during code deploys prevents environment cache invalidation

Previously, catalog compilations for a newly created environment directory could fail if the environment was listed while the directory was being created. This issue only occurred when using an `environment_timeout` value greater than 0 and less than unlimited. This is now fixed. [PUP-10942](#)

Syntax error in previously valid Puppet code due to removal of keywords

The `application`, `consumes`, `produces` and `site application` orchestration keywords were previously removed from the reserved keywords list, causing syntax errors in Puppet code. This is now fixed. [PUP-10929](#)

Retrieve SID for users under `APPLICATION PACKAGE AUTHORITY`

A known issue with `LookupAccountNameW` caused Puppet to fail when managing Windows users under `APPLICATION PACKAGE AUTHORITY` with fully qualified names. This is now fixed and an account name sanitization step has been added to prevent faulty queries. [PUP-10899](#)

Retrieving the current user with the fully-qualified username fails on Windows

Previously, retrieving the current username SID on Windows caused Puppet to fail in certain scenarios, for example, when the user was a secondary domain controller. This release adds a fallback mechanism that uses the fully qualified domain name for lookup. [PUP-10898](#)

Puppet 7.4.1

Released 16 February 2021.

Resolved issues

Puppet users with `forcelocal` are no longer idempotent

This release fixes a regression where setting the `gid` parameter on a user resource with `forcelocal` was not idempotent. [PUP-10896](#)

Puppet 7.4.0

Released 9 February 2021.

New features

New `--timing` option in `puppet facts show`

This release adds a `--timing` option in the `puppet facts show` command. This flag shows you how much time it takes to resolve each fact. [PUP-10858](#)

Resolved issues

User resource with `forcelocal` uses `getent` for groups

The `useradd` provider now checks the `forcelocal` parameter and gets local information on the groups (from `/etc/groups`) and `gid` (from `/etc/passwd`) of the user when requested. [PUP-10857](#)

Slow Puppet agent run after upgrade to version 6

This release improves the performance of the `apt` package provider when removing packages by reducing the calls to `apt-mark showmanual`. [PUP-10856](#)

The `apt` provider does not work with local packages

The `apt` package provider now allows you to install packages from a local file using `source` parameter. [PUP-10854](#)

The `puppet facts show --value-only` command displays a quoted value

Previously, the `puppet facts show --value-only <fact>` command emitted the value as a JSON string, which included quotes around the value, such as `{ "RedHat" }`. It now only emits the value. [PUP-10861](#)

Puppet 7.3.0

Released 20 January 2021.

New features

The `serverport` setting

The `serverport` setting is an alias for `masterport`. [PUP-10725](#)

Enhancements

Multiple `logdest` locations in `puppet.conf` accepted

You can set multiple `logdest` locations using a comma separated list. For example: `/path/file1,console,/path/file2`. [PUP-10795](#)

The `puppet module install` command lists unsatisfiable dependencies

If the `puppet module install` command fails, Puppet returns a more detailed error, including the unsatisfiable module(s) and its ranges. [PUP-9176](#)

New `--no-legacy` option to disable legacy facts

By default, `puppet facts show` displays all facts, including legacy facts. This release adds a `--no-legacy` option to disable legacy facts when querying all facts. [PUP-10850](#)

Resolved issues

The `puppet apply` command creates warnings

This release eliminates Ruby 2.7.x warnings when running `puppet apply` with node statements. [PUP-10845](#)

Remove `Pathname#cleanpath` workaround

This release removes an unnecessary workaround when cleaning file paths, as Ruby 1.9 is no longer supported. [PUP-10840](#)

The `allow *` error message shown during PE upgrade

Puppet no longer prints an error if `fileservers.conf` contains `allow *` rules. It continues to print an error for all other rules, as Puppet's legacy authorization is no longer supported and is superseded by Puppetserver's authorization. [PUP-10851](#)

3x functions cannot be called from deferred functions in Puppet agent

This release allows deferred 3.x functions, like `sprintf`, to be called during a Puppet agent run. [PUP-10819](#)

Cached catalog contains the result of deferred evaluation instead of the deferred function

Puppet 6.12.0 introduced a regression that caused the result of a deferred function to be stored in the cached catalog. As a result, an agent running with a cached catalog would not re-evaluate the deferred function. This is now fixed. [PUP-10818](#)

`puppet facts show` fact output differs from `facter fact`

The output format is different between Facter and Puppet facts when a query for a single fact is provided. This is now fixed. [PUP-10847](#)

Issue with Puppet creating production folder when multiple environment paths are set

Previously, the `production` environment folder was automatically created at every Puppet ran in the first search path, if it did not already exist. This release ensures Puppet searches all the given paths before creating a new `production` environment folder. [PUP-10842](#)

Puppet 7.2.0

This version of Puppet was never released.

Puppet 7.1.0

Released 15 December 2020.

Enhancements

Reduced query time for system user groups

The time it takes to query groups of a system user has been reduced on Linux operating systems with FFI. The `getgrouplist` method is also available. [PUP-10774](#)

Log rotation for Windows based platforms

You can now configure the `pxp-agent` to use the Windows Event Log service by setting the `logfile` value to `eventlog`. [PA-3492](#)

Log rotation for macOS based platforms

This release enables log rotation for the `pxp-agent` on OSX platforms. [PA-3491](#)

Added `server` alias for `routes.yaml`

When `routes.yaml` is parsed, it accepts either `server` or `master` applications. [PUP-10773](#)

OpenSSL bumped to 1.1.1i

This release bumps OpenSSL to 1.1.1i. [PA-3513](#)

Curl bumped to 7.74.0

This release bumps Curl to 7.74.0. [PA-3512](#)

Resolved issues**The Puppet 7 gem is missing runtime dependency on `scanf`**

This is fixed and you can now run module tests against the Puppet gem on Ruby 2.7. [PUP-10797](#)

The `puppet node clean` action LoggerIO needs to implement `warn`

In Puppet 7.0.0, the `puppet node clean` action failed if you had `cadir` in the legacy location or inside the `ssldir`. This was a regression and is now fixed. [PUP-10786](#)

Calling `scope#tags` results in undefined method

Previously, calling the `tags` method within an ERB template resulted in a confusing error message. The error message now makes it clear that this method is not supported. [PUP-10779](#)

User resource is not idempotent on AIX

The AIX user resource now allows for `password` lines with arbitrary whitespace in the `passwd` file. [PUP-10778](#)

Fine grained environment timeout issues

Previously, if the `environment.conf` for an environment was updated and the environment was cleared, `puppetserver` used old values for per-environment settings. This happened if the environment timed out or if the environment was explicitly cleared using `puppetserver`'s environment cache REST API. With this fix, if an environment is cleared, Puppet reloads the per-environment settings from the updated `environment.conf`. [PUP-10713](#)

FIPS compliant nodes are returning an error

This release fixes an issue on Windows FIPS where Leatherman libraries loaded at the predefined address of the OpenSSL library. This caused the OpenSSL library to relocate to a different address, failing the FIPS validation. This is fixed and leatherman compiled with `dynamicbase` is disabled on Windows. [PA-3474](#)

User provider with `uid/gid` as Integer raises warning

This release fixes a warning introduced in Ruby 2.7 that checked invalid objects (such as Integer) against a regular expression. [PUP-10790](#)

Puppet 7.0.0

Released 19 November 2020.

For a list of major changes, see [What's new since Puppet 7](#).

New features

The `puppet facts show` command

You can use the `puppet facts show` command to retrieve a list of facts. By default, it does not return legacy facts, but you can enable it to with the `--show legacy` option. This command replaces `puppet facts find` as the default Puppet facts action. [PUP-10644](#) and [PUP-10715](#)

JSON terminus for node and report

This release implements JSON termini for node and report indirection. The format of the `last_run_report.yaml` report can be affected by the `cache` setting key of the report terminus in the `routes.yaml` file. To ensure the file extension matches the content, update the `lastrunreport` configuration to reflect the terminus changes (`lastrunreport = $statedir/last_run_report.json`). [PUP-10712](#)

JSON terminus for facts

This release adds a new JSON terminus for facts, allowing them to be stored and loaded as JSON. Puppet agents continue to default to YAML, but you can use JSON by configuring the agent application in `routes.yaml`. Puppet Server 7 also caches facts as JSON instead of YAML by default. You can re-enable the old YAML terminus in `routes.yaml`. [PUP-10656](#)

Public folder

There is a new folder with 0755 access rights named `public`, which is now the default location for the `last_run_summary.yaml` report. It has 640 file permissions. This makes it possible for a non-privileged process to read the file. To relax permissions on the last run summary, set the `group` permission on the file in `puppet.conf` to the following [PUP-10627](#): `lastrunsummary = $publicdir/last_run_summary.yaml { owner = root, group = monitoring, mode = 0640 }`

The `settings_catalog` setting

To load Puppet more quickly, you can set the `settings_catalog` setting to `false` to skip applying the settings catalog. The setting defaults to `true`. [PUP-8682](#)

New numeric and port setting types

This release adds a new `port` setting type, which turns the given value to an integer, and validates it if the value is in the range of 0-65535. Puppet port can use this setting type. [PUP-10711](#)

MSI `PUPPET_SERVER` and alias

This release adds a new Windows Installer property called `PUPPET_SERVER`. You can use this as an alias to the existing `PUPPET_MASTER_SERVER` property. [PA-3440](#)

New GPG signing key

Puppet has a new GPG signing key. See [verify packages](#) for the new key.

Enhancements

Ruby version bumped to 2.7

The default version of Ruby is now 2.7. The minimum Ruby version required to run Puppet 7 is now 2.5. After upgrading to Puppet 7, you may need to use the `puppet_gem` provider to ensure all your gems are installed. [PUP-10625](#)

Default digest algorithm changed to sha256

Puppet 7 now uses sha256 as the default digest algorithm. [PUP-10583](#)

Gem provider installs gems in Ruby

The gem provider now installs gems in Ruby by default. Use the `puppet_gem` provider to reinstall gems in the Ruby distribution vendored in Puppet. For example, if custom providers or deferred functions require gems during catalog application. [PUP-10677](#)

FFI functions, structs and constants moved to a separate Windows module

To increase speed, we have moved FFI functions, constants and structures out of `Puppet::Util::Windows`. [PUP-10606](#)

Default value of `ignore_plugin_errors` changed from true to false

The default value for `ignore_plugin_errors` is now false. This stops Puppet agents failing to `pluginsync`. [PUP-10598](#)

Interpolation of sensitive values in EPP templates

Previously, if you interpolated a sensitive value in a template, you were required to unwrap the sensitive value and rewrap the result. Now the `epp` and `inline_epp` functions automatically return a `Sensitive` value if any interpolated variables are sensitive. For example: `inline_epp("Password is <%= Sensitive('opensesame') %>")`. Note that these changes just apply to EPP templates, not ERB templates. [PUP-8969](#)

`sshkeys_core` module bumped to 2.2.0

Puppet 7 bumps the `sshkeys_core` modules to 2.2.0 in the Puppet agent. [PA-3473](#)

Call simple server status endpoint

Puppet updates the endpoint for checking the server status to `/status/v1/simple/server`. If the call returns a 404, it makes a new call to `/status/v1/simple/master`, and ensures backwards compatibility. [PUP-10673](#)

Default value of `disable_i18n` changed from false to true

The default value for the `disable_i18n` setting has changed from false to true and locales are not `pluginsync`d when i18n is disabled. [PUP-10610](#)

`Pathspec` no longer vendored

The `pathspec` Ruby library is no longer vendored in Puppet. If you require this functionality, you need to install the `pathspec` Ruby gem. [PUP-10107](#)

Deprecations and removals**`func3x_check` setting removed**

The `func3x_check` setting has been removed. [PUP-10724](#)

`master_used` report parameter removed

The deprecated `master_used` parameter has been removed. Instead use `server_used`. [PUP-10714](#)

facterng feature flag removed

The `facterng` feature flag has been removed. It is not needed anymore as Puppet 7 uses `Facter 4` by default. [PUP-10605](#)

held removed from apt provider

The `apt` provider no longer accepts deprecated `ensure=held`. Use the `mark` attribute instead. [PUP-10597](#)

Method from DirectoryService removed

The deprecated `DirectoryService#write_to_file` method has been removed. [PUP-10489](#)

Method from Puppet::Provider::NameService removed

The deprecated `Puppet::Provider::NameService#listbyname` method has been removed. [PUP-10488](#)

Methods from TypeCalculator removed

The deprecated `TypeCalculator.enumerable` has been removed, and the functionality has been moved to `Iterable`. [PUP-10487](#)

Enumeration type removed

The deprecated `Enumeration` class has been removed, and its functionality has been moved to `Iterable`. [PUP-10486](#)

Puppet::Util::Yaml.load_file removed

The deprecated `Puppet::Util::Yaml.load_file` method has been removed. [PUP-10475](#)

Puppet::Resource methods removed

The following deprecated `Puppet::Resource` methods have been removed:

- `Puppet::Resource.set_default_parameters`
- `Puppet::Resource.validate_complete`
- `Puppet::Resource::Type.assign_parameter_values`. [PUP-10474](#)

legacy auth.conf support removed

The legacy `auth.conf` has been deprecated for several major releases. Puppet 7 removes all support for legacy `auth.conf`. Instead, authorization to Puppet REST APIs is controlled by `puppetserver auth.conf`. In addition, the `allow` and `deny` rules in `fileserver.conf` are now ignored and Puppet logs an error for each entry. The `rest_authconfig` setting has also been removed. [PUP-10473](#)

Puppet.define_settings removed

The deprecated `Puppet.define_settings` method has been removed. [PUP-10472](#)

Application orchestration language features removed

The deprecated application orchestration language features have been removed. The keywords `application`, `site`, `consumes` and `produces`, and the `export` and `consume` metaparameters, now raise errors. The keywords are still reserved, but can't be used as a custom resource type or attribute name. The environment catalog REST API has also been removed, along with supporting classes, such as the environment compiler and validators. [PUP-10446](#)

Puppet::Network::HTTP::ConnectionAdapter removed

The `Puppet::Network::HTTP::ConnectionAdapter` has been removed, and contains the following breaking changes:

- The Client networking code has been moved to `Puppet::HTTP`.
- The `Puppet::Network::HttpPool.http_instance` method has been removed.
- The `Puppet.lookup(:http_pool)` has been removed.
- The deprecated `Puppet::Network::HttpPool.http_instance` and `connection` methods have been preserved. [PUP-10439](#)

environment_timeout_mode setting removed

The `environment_timeout_mode` setting has been removed. Puppet no longer supports environment timeouts based on when the environment was created. In Puppet 7, the `environment_timeout` setting is always interpreted as 0 (never cache), unlimited (always cache), or from when the environment was last used. [PUP-10619](#)

Networking code from the parent REST terminus removed

The Networking code from the parent REST terminus has been removed, and is a breaking change for any REST terminus that relies on the parent REST terminus to perform the network request and process the response. The REST terminus must implement the `find`, `search`, `save` and `destroy` methods for their indirected model. [PUP-10440](#)

Dependency on http-client gem removed

The dependency on the `http-client` gem has been removed. If you have a Puppet provider that relies on this gem, you must install it. [PUP-10490](#)

HTTP file content terminus removed

The HTTP file content terminus has been removed. It is no longer possible to retrieve HTTP file content using the indirector. Instead, use Puppet's builtin HTTP client instead: `response = Puppet.runtime[:http].get(URI("http://example.com/path"))`. [PUP-10442](#)

Puppet::Util::HttpProxy.request_with_redirects removed

The `Puppet::Util::HttpProxy.request_with_redirects` method has been removed, and moves the `Puppet::Util::HttpProxy` class to `Puppet::HTTP::Proxy`. The old constant is backwards compatible. [PUP-10441](#)

Puppet::Rest removed

`Puppet::Rest` removed and `Puppet::Network::HTTP::Compression` have been removed. This change moves `Puppet::Network::Resolver` to `Puppet::HTTP::DNS` and deprecates `Puppet::Network::HttpPool` methods. [PUP-10438](#)

Remove strict_hostname_checking removed

The deprecated `strict_hostname_checking` and `node_name` settings have been removed. The functionality of these settings is possible using explicit constructs within a `site.pp` or fully featured enc. [PUP-10436](#)

puppet module build, generate and search actions removed

The `puppet module build`, `generate` and `search` actions have been removed. Use Puppet Development Kit (PDK) instead. [PUP-10387](#)

puppet status application has been removed

The deprecated `puppet status` application has been removed. [PUP-10386](#)

The puppet cert and key commands removed

The non-functioning `puppet cert` and `puppet key` commands have been removed. Instead use `puppet ssl` on the agent node and `puppetserver ca` on the CA server. [PUP-10369](#)

SSL code, termini and settings removed

The following SSL code, termini and settings have been removed:

- `Puppet::SSL::Host`
- `Puppet::SSL::Key`
- `Puppet::SSL::{Certificate,CertificateRequest}.indirection`
- `Puppet::SSL::Validator*`
- `ssl_client_ca_auth`
- `ssl_server_ca_auth` [PUP-10252](#)

The func3x_check setting has been removed

The setting to turn off `func 3x` API validation has been removed. Now all 3x functions are validated. [PUP-9469](#)

The future_features logic has been removed

The unused `future_features` setting has been removed. [PUP-9426](#)

The puppet man application has been removed

The `puppet man` application is no longer needed and has been removed. The agent package now installs man pages so that `man puppet` produces useful results. Puppet's help system (`puppet help`) is also available. [PUP-8446](#)

The execfail method from util/execution has been removed

The following deprecated methods have been removed:

- `Puppet::Provider#execfail`
- `Puppet::Util::Execution.execfail`. [PUP-7584](#)

The win32-process has been removed

The Puppet dependency on the `win32-process` gem has been removed. You can implement the functionality using FFI. [PUP-7445](#)

The win32-service gem has been removed

The dependency on the `win32-service` gem has been removed and uses the `Daemon` class in Puppet instead. [PUP-5758](#)

The win32-security gem has been removed from Puppet

To improve Puppet's handling of Unicode user and group names on Windows, some of the code interacting with the Windows API has been rewritten to ensure wide character (UTF-16LE) API variants are called. As a result, Puppet no longer needs the `win32-security` gem. Any code based references to the gem have been removed. The gem currently remains for backward compatibility, but is to be removed in a future release. [PUP-5735](#)

The capability to install an agent on Windows 2008 and 2008 R2 has been removed

You can no longer install Puppet 7 agents on Windows versions lower than 2012. [PA-3364](#)

Support for Ruby versions older than 2.5 removed

Support for Ruby versions older than 2.5 has been removed, and Fixnum and Bignum have been replaced with Integer. [PUP-10509](#)

`dir monkey-patch` removed

This external dependency on the win32/dir gem has been removed and replaces CSIDL constants with environment variables. [PUP-10653](#)

Master removed from docs

Documentation for this release replaces the term master with primary server. This change is part of a company-wide effort to remove harmful terminology from our products. For the immediate future, you'll continue to encounter master within the product, for example in parameters, commands, and preconfigured node groups. Where documentation references these codified product elements, we've left the term as-is. As a result of this update, if you've bookmarked or linked to specific sections of a docs page that include master in the URL, you'll need to update your link.

Resolved issues

Puppet agent installation fails when `msgpack` is enabled on `puppetserver`

Previously, the agent failed to deserialize the catalog and fail the run if the `msgpack` gem was enabled but not installed. Now the agent only supports that format when the `msgpack` gem is installed in the agents vendored Ruby. [PUP-10772](#)

Puppet feature detection leaves Ruby gems in a bad state

This release fixes a Ruby gem caching issue that prevented the agent from applying a catalog if a gem was managed using the native package manager, such as yum or apt. [PUP-10719](#)

Puppet 6 agents do not honor the `usecacheonfailure` setting when using `server_list`

Previously, when `server_list` was used when there was no server accessible, the Puppet run failed even if `usecacheonfailure` was set to true. Now Puppet only fails if `usecacheonfailure` is set to false. [PUP-10648](#)

Setting `certname` in multiple sections bypasses validation

Previously, Puppet only validated the `certname` setting when specified in the main setting, but not if the value was in a non-global setting like agent. As a result, it was possible to set the `certname` setting to a value containing uppercase letters and prevent the agent from obtaining a certificate the next time it ran. Puppet now validates the `certname` setting regardless of which setting the value is specified in. [PUP-9481](#)

Issues caused by backup to the local `filebucket`

By default, Puppet won't backup files it overwrites or deletes to the local `filebucket`, due to issues where it became unbounded. You can re-enable the local `filebucket` by setting `File { backup => 'puppet' }` as a resource default. [PUP-9407](#)

Remove future feature flag for `prefetch_failed_providers` in `transaction.rb`

If a provider `prefetch` method raises a `LoadError` or `StandardError`, the resources associated with the provider are marked as failed, but unrelated resources are applied. Previously this behavior was controlled by the `future_features` flag, and disabled by default. [PUP-9405](#)

Change default value of `hostcsr` setting

The default value of the `hostcsr` setting has been updated to match where Puppet stores the certificate request (CSR) when waiting for the CA to issue a certificate. [PUP-9346](#)

Refactor the SMF provider to implement enableable semantics

Previously, the SMF provider did not properly implement enableable semantics. Now `enable` and `ensure` are independent operations where `enable` handles whether a service starts or stops at boot time, and `ensure` handles whether a service starts or stops in the current running instance. [PUP-9051](#)

The list of reserved type names known to the parser validator is incomplete

A class or defined type in top scope can no longer be named `init`, `object`, `sensitive`, `semver`, `semverrange`, `string`, `timestamp`, `timespan` or `typeset`. You can continue to use these names in other scopes such as `mymodule::object`. [PUP-7843](#)

Export or virtualize class error

Previously, Puppet returned a warning or error if it encountered a virtual class or an exported class, but it still included resources from the virtual class in the catalog. Now Puppet always error on virtual and exported classes. [PUP-7582](#)

`Puppet::Util::Windows::String.wide_string` embeds a NULL char

This release removes a Ruby workaround for wide character strings on Windows. [PUP-3970](#)

`puppet config set certname` accepts upper-case names

Previously, the `puppet config set` command could set a value that was invalid, causing Puppet to fail the next time it ran or the service was restarted. Now the command validates the value before committing the change to `puppet.conf`. [PUP-2173](#)

Unable to read `last_run_summary.yaml` from user

Puppet agent code now aligns with the new `last_run_summary.yaml` location. [PA-3253](#)

Puppet known issues

These are the known issues in this version of Puppet.

User and group management on macOS 10.14 and above requires Full Disk Access (FDA)

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

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

To give Puppet access, go to **System Preferences > Security & Privacy > Privacy > Full Disk Access**, and add the path to the Puppet executable, along with any other parent processes you use to run. For detailed steps, see [Add](#)

[full disk access for Puppet on macOS 10.14 and newer](#) on page 47. Alternatively, set up automatic access using Privacy Preferences Control Profiles and a Mobile Device Management Server. [PA-2226](#), [PA-2227](#)

The `puppet node clean` command fails for users who have their `cadir` in the legacy location

In Puppet 7, the default location of the `cadir` has moved. If you have it in the old location, most upgrades trigger a warning when executing commands from Puppet. It causes the `puppet node clean` command to fail. [PUP-10786](#)

Hiera `knockout_prefix` is ineffective in hierarchies more than three levels deep

When specifying a deep merge behaviour in Hiera, the `knockout_prefix` identifier is effective only against values in an adjacent array, and not in hierarchies more than three levels deep. [HI-223](#)

Specify the epoch when using version ranges with the `yum` package provider

When using version ranges with the `yum` package provider, there is a limitation which requires you to specify the epoch as part of the version in the range, otherwise it uses the implicit epoch ``0``. For more information, see the [RPM packaging guide](#). [PUP-10298](#)

Deferred functions can only use built-in Puppet types

Deferred functions can only use types that are built into Puppet (for example `String`). They cannot use types from modules like `stdlib` because Puppet does not plugin-sync these types to the agent. [PUP-8600](#)

The Puppet agent installer fails when `systemd` is not present on Debian 9

The `puppet-agent` package does not include `sysv` init scripts for Debian 9 (Stretch) and newer. If you have disabled or removed `systemd`, `puppet-agent` installation and Puppet agent runs can fail.

Upgrading Windows agent fails with `ScriptHalted` error

Registry references to `nssm.exe` were removed in [PA-3263](#). Upgrading from a version without this update to a version that contains it triggers a Windows `SecureRepair` sequence that fails if any of the files delivered in the original `*.msi` package are missing. This is an issue when upgrading to one of the following Puppet agent versions: 5.5.21, 5.5.22, 6.17.0, 6.18.0, 6.19.0, 6.19.1, 6.20.0, 7.0.0, 7.1.0 or 7.3.0. To work around this issue, put the `*.msi` file for the currently installed version in the `C:\Windows\Installer` folder before you upgrade. Starting with Puppet agent 6.21.0 and 7.4.0, the `nssm.exe` registry value will be replaced with an empty string, instead of the registry key being removed, to avoid triggering Windows `SecureRepair`. [PA-3545](#)

The Puppet agent installer fails when `systemd` is not present on Debian 9

In versions less than 7.4.0, the `puppet-agent` package does not include `sysv` init scripts for Debian 9 (Stretch) and newer. If you had disabled or removed `systemd`, the `puppet-agent` installation and agent runs could fail. This is now fixed. [PA-2028](#)

Facter release notes

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

Facter 4.0.52

Released 16 March 2021 and shipped with Puppet 7.5.0.

New features

- **Azure metadata fact.** This release adds the `az_metadata` fact which provides information on Azure virtual machine instances. For more information, see the [Microsoft Azure instance metadata documentation](#). [FACT-1383](#)

- **Azure identification fact.** This release adds the `cloud.provider` fact for Azure identification on Linux and Windows platforms. [FACT-1847](#)

Enhancements

- **Dependency to `lsb` packages are missing** The `os.distro` facts are now resolved without the `lsb_release` on the following platforms: RHEL, Amazon, SLES. The information is read from the system in the same way. Note that the `os.distro.specification` fact, which refers to `lsb` version, is available only if the `lsb_release` has been installed. [FACT-2931](#)
- **Linux networking resolver split into four classes.** The Linux networking resolver was too large and has now been split into four classes. The classes are: `toSocketParser` (gets data from the Ruby Socket library), `DHCP` (gets all DHCP related data), `RoutingTable` (gets interface data from the `ip route show` command, if something can not be retrieved with `SocketParser`) and the Linux resolver (combines the data from the other classes). [FACT-2915](#)

Resolved issues

- **Permission denied error when reading Facter cache during PE 2021 upgrade.** Facter 4 no longer loads `facter.conf` from the default location when running on jRuby. [FACT-2959](#)
- **Facter 4 reports `lsbmajdistrelease` on Ubuntu differently from Facter 3.** Previously, the `lsbmajdistrelease` fact from Facter 4 was not showing the correct value on Ubuntu. This is now fixed and aligns the fact's output with Facter 3. [FACT-2952](#)
- **Root of structured core facts cannot be overridden by a custom fact.** Previously, the root of structured core facts could not be overridden by a custom fact — because the top-level fact did not exist. This release updates the `QueryParser` logic to return the root fact — if present in the loaded facts list — and allows redefinition of core facts. [FACT-2950](#)
- **Facter tries to load an incompatible `libsocket.so` on SmartOS.** Previously, Facter tried to load an incompatible (32-bit) `libsocket.so` from a hardcoded path using `ffi` and failed to retrieve networking facts. Now, the library name is preferred and networking facts can be retrieved on SmartOS. Contributed by Puppet community member [smokris](#). [FACT-2947](#)
- **Puppet Server creates another certname during upgrade.** Previously, when upgrading Puppet 6 to Puppet 7 on Linux, Facter failed to retrieve the domain using JRuby because the `Socket.getaddrinfo` calls failed. Now, if any of the `Socket` method calls fail, Facter can retrieve the information using FFI methods. [FACT-2944](#)
- **The puppet facts show command logs error when stdlib is installed.** This release fixes an issue where `Facter.value` did not return the fact value for a legacy fact. This happened when calling a legacy fact from a custom fact or calling other Facter API methods before calling `Facter.value`. [FACT-2937](#)
- **Facter 4 `pe-bolt-server` raises access denied error if cache is enabled.** When Facter is unable to delete the cache files, it now logs a warning message instead of returning error. [FACT-2961](#)
- **Facter 4 does not accept the same time units as Facter 3.** This release makes the following `ttls` time units available in Facter 4: `ns`, `nanos`, `nanoseconds`, `us`, `micros`, `microseconds`, `ms`, `milis`, `milliseconds`, `s`, `m`, `h`, `d`. Note that singular time units are also accepted, for example, `mili` and `nano`. [FACT-2962](#)
- **The `Facter.conf` file does not accept singular `ttls` units.** Previously, Facter failed if the `ttls` unit in the `facter.conf` file was a plural ("days" and "hours"). Facter now accepts a singular noun ("day" and "hour").

Facter 4.0.51

Released 16 February 2021.

This release includes minor maintenance changes. For the latest features, see the release notes for Facter 3.14.50.

Factor 4.0.50

Released 9 February 2021 and shipped with Puppet 7.4.0.

Enhancements

- **Networking fact improvements for AIX.** The AIX networking resolver now uses FFI to detect networking interfaces and IPs. Previously, VLANs and secondary IPs were not displayed. [FACT-2878](#)
- **Improved performance for blocking legacy facts.** This release improves the performance of blocking legacy facts by implementing a different mechanism. The new mechanism does not allow legacy groups to be overridden by a group with the same name in `fact-groups`. If you add a legacy group in `fact-groups`, it is ignored. The new implementation is faster for use cases involving multiple custom facts that depend on core facts. [FACT-2917](#)

Resolved issues

- **Factor::Core::Execution does not set status variables in Factor 4.** This release reimplements `Open3.popen3` to use `Process.wait` instead of `Process.detach`. [FACT-2934](#)
- **Factor error message — no implicit conversion of nil into String — when determining processor speed on Linux.** Factor now handles processor speed values where `log10` is not set (3, 6, 9, 12). [FACT-2927](#)
- **Factor fails "closed" if the factor.conf file is invalid.** Previously, Factor failed when an invalid config file was provided. Factor now logs a warning message stating that the parsing of the config file failed and continues retrieving facts with the default options. [FACT-2924](#)
- **Domain on Windows does not prioritise registry.** Factor now prioritises information from registry on Windows, instead of network interface domain names. [FACT-2923](#)
- **LinuxMint Tessa not recognized.** Previously, the `os.release` fact was retrieved from the `/etc/os-release` file, but Factor 3 read other release files based on operating system (OS). Now Factor retrieves `os.release` from the specific release file for every OS. [FACT-2921](#)

Factor 4.0.49

Released 20 January 2021 and shipped with Puppet Platform 7.3.0.

Resolved issues

- **Aggregate facts are broken.** Previously, Factor broke when trying to add a debug message for the location where aggregate facts are resolved from. This only happened with aggregate facts that returned an array or hash without having an aggregate block call. [FACT-2919](#)

Factor 4.0.48

Released 20 January 2021.

Enhancements

- **Rewritten tests for Linux networking resolver.** This release refactors the Linux networking resolver, including fixing the unit tests and adding new ones. [FACT-2901](#)

Resolved issues

- **Factor 4.0.x does not return the domain correctly when set in the registry.** Previously, Factor did not retrieve the domain correctly on Linux and resulted in a faulty FQDN facts. Factor also failed to retrieve domain facts when Windows did not expose the host's primary DNS suffix. This is now fixed. [FACT-2882](#)
- **Legacy group blocks processors core fact.** Blocking legacy facts no longer blocks processors core fact. [FACT-2911](#)
- **Factor Hocon output format.** The `--hocon` option now functions as intended. [FACT-2909](#)

- **Legacy blockdevice vendor and size facts not resolving.** This release fixes `blockdevice_*_size` facts not resolving on AIX, and `blockdevice_*_vendor` facts not resolving on Linux and Solaris. [FACT-2903](#)
- **Facter detects OS family without checking or translating the information.** Previously, Facter detected the OS family by reading `/etc/os-release` without checking or translating the information. This is now fixed and Facter translates the `id_like` field from `/etc/os-release` to Facter known families. [FACT-2902](#)

Facter 4.0.47

Released 15 December 2020 and shipped with Puppet 7.1.0.

New features

- **Added `scope6` fact for per binding.** This release adds the `scope6` fact under every `ipv6` address from the `interface.bindings6` fact. [FACT-2843](#)
- **Support for AWS IMDSv2.** This release updates the EC2 fact to use IMDSv2 to authenticate. To use v2, set the `AWS_IMDSv2` environment variable to `true`. Note that the token is cached for a maximum of 100 seconds.

Resolved issues

- **Facter 4 does not resolve hostname facts.** Previously, Facter failed when `Socket.getaddrinfo` was called, which prevented retrieval of FQDN information. This is now fixed. [FACT-2894](#)
- **Gem-based Facter 4 does not log the facts in debug mode.** This release adds log messages for resolved fact values. [FACT-2883](#)
- **Gem-based Facter 4 does not return the complete FQDN.** Previously, domain was not retrieved correctly on Linux based systems and resulted in a faulty FQDN fact. This release uses Ruby Socket methods to retrieve domain correctly. [FACT-2882](#)
- **Puppet 7 treats non existent facts differently to Puppet 6.** This release excludes custom facts with nil value from `to_user_output`, `values` and `to_hash` Ruby Facter API's. The custom facts with nil value are still returned by `value`, `fact` and `[]` API's. [FACT-2881](#)
- **Facts failing on machines with VLANs.** With this release, `dots` in legacy fact names are ignored. They are not used as an indicator of a fact hierarchy because legacy facts cannot compose and have a flat (key - value) structure. [FACT-2870](#)
- **Facter 4 changes `is_virtual` fact from boolean to string.** This release fixes a regression that caused the `is_virtual` to be a string instead of a boolean. [FACT-2869](#)
- **External facts are loaded when using puppet lookup for a different node.** The `load_external` API method was missing in Facter 4. This is now fixed. [FACT-2859](#)
- **Facter fails when the interface name is not UTF-8.** Previously, a pointer used to indicate networking information was being released by the GC too early and the memory was overridden, resulting in inconsistent data. The fix extends the scope of the pointer so that the memory it points to does not release prematurely. [FACT-2856](#)
- **Failure when a structured custom fact has the wrong layout.** This release adds a log message when custom fact names are incompatible and a fact hierarchy cannot be created. [FACT-2851](#)
- **Cannot retrieve local facts error.** This release fixes an error thrown on systems with a low file descriptor limit. [FACT-2898](#)
- **Dig method fails on Puppet `$facts`.** The Facter 4 API method `to_hash` returned a different data type to Facter 3. This release ensures the `to_hash` method returns a Ruby Hash instance. [FACT-2897](#)
- **Facter fails when trying to retrieve ssh facts.** Facter now skips reading ssh keys it does not recognise. [FACT-2896](#)
- **Missing primary interface check on all platforms.** Similar to Facter 3, this release adds a final check that detects the primary interface from the IP. Localhost IPs are excluded. [FACT-2892](#)
- **Facter 4.0.46 breaks virtual flag.** Facter now checks whether the `/proc/1ve/list` file is a regular file, instead of checking if it is executable. [FACT-2891](#)
- **Facter 4.0.46 does not load external fact files in lexicographical order.** This is now fixed. [FACT-2874](#)

- **Secondary interfaces are not reported.** `Networking.interfaces` now display secondary interfaces (with or without the label) and the `VLANs.MAC` address is correctly displayed for bonded interfaces. If DHCP is not found, use the `dhcpcd -U <interface_name>` command to search. [FACT-2872](#)

Deprecations and removals

Update rake task for generating facts and tests. The scripts used to generate facts were outdated and had never been used. This release removes them. [FACT-2298](#)

Factor 4.0.46

Released 19 November 2020 and shipped with Puppet 7.0.0.

New features

- **Operating system hierarchy.** Factor 4 introduces a hierarchy for operating systems. The hierarchy allows you to load facts from the child and all parents. If the same fact is present in a child and a parent, the one from the child takes precedence. [FACT-2555](#)
- **Rake task for mapping fact name and fact class.** To improve the visibility of which facts are loaded for an operating system, Factor 4 has a rake task that prints all facts and the class that resolved that fact. [FACT-2557](#)
- **Blocklist.** Factor 4 allows you to block facts at a granular level — you can block any fact from the fact hierarchy, for both groups of facts and individual facts. [FACT-1976](#)
- **Block legacy facts.** Legacy facts are a subtype of core facts and you can now block them, like any core fact, using the `blocklist` from `factor.conf`. [FACT-2259](#)
- **Block custom facts.** Factor 4 allows you to block custom facts. You can add the fact to the `blocklist` from `factor.conf`. [FACT-2718](#)
- **Block external facts.** Factor 4 allows you to block external facts. Blocking external facts is different from blocking core and custom facts — you need to specify the name of the file from which external facts are loaded to `blocklist` in `factor.conf`. [FACT-2717](#)
- **The puppet facts show command.** The `factor -p` and `factor --puppet` commands have been replaced with `puppet facts show`. [FACT-2719](#)
- **Group for legacy facts.** The legacy group contains all the legacy facts. You can find it in `lib/factor/config.rb` and can block it in `factor.conf`. [FACT-2296](#)
- **The fact-groups group.** You can define your own custom groups in `factor.conf` using the new `fact-groups` group. The `blocklist` and `ttls` groups from `factor.conf` accept predefined groups, custom groups or fact names. You must use the file name when using external facts. [FACT-2331](#)
- **Define fact groups for blocking or caching.** You can define new fact groups in `factor.conf`, and use the group to block or cache facts. [FACT-2515](#)
- **External fact caching.** To cache external facts, use the filename of the external fact when setting the `ttls` in `factor.conf`. [FACT-2619](#)

Enhancements

- **CLI compatible with Factor 3.** Factor 4 reimplemented the Factor 3 command line interface using `thor` gem. [FACT-1955](#)
- **Loaded facts based on operating system hierarchy.** To improve performance, Factor 4 only loads the files and facts that are needed for the operating system it is running on. [FACT-2093](#)
- **Log class name in log messages.** Improved logging in Factor 4 prints the class from which the log was generated. [FACT-2036](#)
- **Restored --timing option to native factor.** The restored `--timing` and `-t` arguments allow you to see how much time it took each fact to resolve. [FACT-1380](#)
- **(Experimental). Reverted parallel resolution of facts.** To improve performance, facts are resolved in parallel on JRuby. [FACT-2819](#)
- **Timeout on resolution.** You can now specify the `timeout` attribute in custom fact options. [FACT-2643](#)

- **Caching core facts** . To cache core facts, add the fact group to `factor.conf` `ttls`. Fact values are stored and retrieved on future runs. After the `ttls` expires, the fact is refreshed. [FACT-2486](#)

Resolved issues

- **Fix Ruby 2.7 warning on Factor 4**. Factor 4 now supports Ruby 2.7. [FACT-2649](#)
- **Factor does not support timeout for shell calls**. External commands have a timeout, and if they do not complete in the given time, they are forced stop. The default timeout is 300 seconds. You can now specify a timeout using the `limit` attribute in `Factor::Core::Execution.execute`. [FACT-2793](#)
- **Fact names are treated as regex and can lead to caching of unwanted facts**. The regex used to detect facts has been improved to distinguish between fact groups and legacy facts. [FACT-2787](#)
- **Factor uptime shows host uptime inside docker container**. Previously, the kernel only reported the host uptime inside a Docker container. You can now see the container uptime. [FACT-2737](#)
- **A fact present in two groups does not get cached if the second groups has a ttls**. If a fact is present in two groups, and both of them have a `ttls` defined in `factor.conf`, the lowest `ttls` is takes precedence. [FACT-2786](#)

Factor known issues

These are the known issues in this version of Factor.

The implementation of the dot notation is not compatible with the Puppet ecosystem

The implementation of the [dot notation feature](#) in Factor 4 is not compatible with other parts of the Puppet ecosystem, and does not give the fact the same name as Factor 3. Factor 4.1 will revert back to the way Factor 3 handled facts with dotted names. [FACT-3004](#)

Incorrect OS description output on Debian 9

Running the `factor` command in Factor 4 returns incorrect output for the `os.distro.description` fact on Debian 9. These facts work correctly in Factor 3 and appear differently when running the `puppet facts diff` command in Factor 4. [FACT-2963](#)

Puppet Server release notes

Puppet Server 7.1.0

Released 16 March 2021.

Enhancements

- Puppet Server now adds an extension for subject-alternative-name (SAN) when it signs incoming certificate signing requests (CSR). The SAN extension contains the common name (CN) as a dns-name on the certificate. If the CSR comes with its own SAN extension, Puppet Server signs it and ensures the SAN extension includes the CSR's CN. [SERVER-2338](#)

Bug fixes

- The Jetty webserver now uses the local copy of the CRL from Puppet's SSL directory instead of the CA's copy. This fix makes it easier to set up compilers, which always have a disabled CA service and no CRL at the CA path. [SERVER-2558](#)

Deprecations

- The `master-conf-dir`, `master-code-dir`, `master-var-dir`, `master-log-dir`, and `master-run-dir` configuration settings have been deprecated in favor of `server-conf-dir`, `server-code-dir`,

`server-var-dir`, `server-log-dir`, and `server-run-dir` respectively. The configuration files — which use the new settings — are shipped with the 7.1.0 `puppetserver` package. Note that the old settings are still honored for backwards compatibility, but we recommend you upgrade to the new settings. [SERVER-2867](#)

Puppet Server 7.0.3

Released 9 February 2021.

This release updates dependencies to include security fixes.

Puppet Server 7.0.2

Released 20 January 2021

Bug fix

- The warning issued when the CA dir is inside the SSL dir now only prints server logs at startup and when using the `puppetserver ca` CLI, instead of any time a Puppet command is used. ([SERVER-2934](#))

Puppet Server 7.0.1

Released 15 December 2020.

Enhancements

- The JRuby version has been bumped from 9.2.13.0 to 9.2.14.0. ([SERVER-2925](#))

Bug fixes

- The CA command line tool now correctly honors the `server` sections in the `puppet.conf`.
- When creating the symlink between the new and legacy cads the symlink will now be properly owned by the `puppet` user. ([SERVER-2917](#))

Puppet Server 7.0.0

Released 17 November 2020.

Puppet Server 7.0 is a major release. It breaks compatibility with agents prior to 4.0 and the legacy Puppet `auth.conf`, moves the default location for the `cadir`, and changes defaults for fact caching and cipher suites. See below for more details. Caution is advised when upgrading.

New features

- The default value for the `cadir` setting is now located at `/etc/puppetlabs/puppetserver/ca`. Previously, the default location was inside Puppet's own `ssldir` at `/etc/puppetlabs/puppet/ssl/ca`. This change makes it safer to delete Puppet's `ssldir` without accidentally deleting your CA certificates.
- The `puppetserver` CA CLI now provides a `migrate` command to move the CA directory from the Puppet `confdir` to the `puppetserver confdir`. It leaves behind a symlink on the old CA location, pointing to the new location at `/etc/puppetlabs/puppetserver/ca`. The symlink provides backwards compatibility for tools still expecting the `cadir` to exist in the old location. In a future release, the `cadir` setting will be removed entirely. ([SERVER-2896](#))
- The default value for the facts cache is now JSON instead of YAML. You can re-enable the old YAML terminus in `routes.yaml`. ([PUP-10656](#))
- Support for legacy Puppet `auth.conf` has been removed and the `jrubby-puppet.use-legacy-auth-conf` setting no longer works. Use Puppet Server's `auth.conf` file instead. ([SERVER-2778](#))
- Puppet Server no longer services requests for legacy (3.x) Puppet endpoints. Puppet Agents before 4.0 are no longer be able to check in. ([SERVER-2791](#))
- This release removes default support for many cipher suites when contacting Puppet Server. The new default supported cipher suites are: `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256`, `TLS_DHE_RSA_WITH_AES_256_GCM_SHA384`, `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`, `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`,

TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256, and TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384. This change aligns open source Puppet with Puppet Enterprise. Note that this change may break on old platforms. To re-enable older cipher suites you may edit the `webserver.conf`. Valid cipher suite names are listed in the [JDK Documentation](#). (SERVER-2913)

- Puppet Server now provides an HTTP client whose API conforms to the HTTP client provided by Puppet. This new client is stored in the Puppet runtime as `Puppet.runtime[:http]`. (SERVER-2780)

Puppet Server known Issues

For a list of all known issues, visit our [Issue Tracker](#).

Cipher updates in Puppet Server 6.5

Puppet Server 6.5 includes an upgrade to the latest release of Jetty's 9.4 series. With this update, you may see "weak cipher" warnings about ciphers that were previously enabled by default. Puppet Server now defaults to stronger FIPS-compliant ciphers, but you must first remove the weak ciphers.

The ciphers previously enabled by default have not been changed, but are considered weak by the updated standards. Remove the weak ciphers by removing the `cipher-suite` configuration section from the `webserver.conf`. After you remove the `cipher-suite`, Puppet Server uses the FIPS-compliant ciphers instead. This release includes the weak ciphers for backward compatibility only.

The FIPS-compliant cipher suites, which are not considered weak, will be the default in a future version of Puppet. To maintain backwards compatibility, Puppet Server explicitly enables all cipher suites that were available as of Puppet Server 6.0. When you upgrade to Puppet Server 6.5.0, this affects you in two ways:

1. The 6.5 package updates the `webserver.conf` file in Puppet Server's `conf.d` directory.
2. When Puppet Server starts or reloads, Jetty warns about weak cipher suites being enabled.

This update also removes the `so-linger-seconds` configuration setting. This setting is now ignored and a warning is issued if it is set. See Jetty's [so-linger-seconds](#) for removal details.

Note: On some older operating systems, you might see additional warnings that newer cipher suites are unavailable. In this case, manage the contents of the `webserver.cipher-suites` configuration value to be those strong suites that available to you.

Server-side Ruby gems might need to be updated for upgrading from JRuby 1.7

When upgrading from Puppet Server 5 using JRuby 1.7 (9k was optional in those releases), Server-side gems that were installed manually with the `puppetserver gem` command or using the `puppetserver_gem` package provider might need to be updated to work with the newer JRuby. In most cases gems do not have APIs that break when upgrading from the Ruby versions implemented between JRuby 1.7 and JRuby 9k, so there might be no necessary updates. However, two notable exceptions are that the `autosign` gem should be 0.1.3 or later and `yard-doc` must be 0.9 or later.

Potential JAVA ARGS settings

If you're working outside of lab environment, increase `ReservedCodeCache` to 512m under normal load. If you're working with 6-12 JRuby instances (or a `max-requests-per-instance` value significantly less than 100k), run with a `ReservedCodeCache` of 1G. Twelve or more JRuby instances in a single server might require 2G or more.

Similar caveats regarding scaling `ReservedCodeCache` might apply if users are managing `MaxMetaspace`.

tmp directory mounted noexec

In some cases (especially for RHEL 7 installations) if the `/tmp` directory is mounted as `noexec`, Puppet Server may fail to run correctly, and you may see an error in the Puppet Server logs similar to the following:

```
Nov 12 17:46:12 fqdn.com java[56495]: Failed to load feature test for posix:
  can't find user for 0
Nov 12 17:46:12 fqdn.com java[56495]: Cannot run on Microsoft Windows
  without the win32-process, win32-dir and win32-service gems: Win32API only
  supported on win32
Nov 12 17:46:12 fqdn.com java[56495]: Puppet::Error: Cannot determine basic
  system flavour
```

This is caused by the fact that JRuby contains some embedded files which need to be copied somewhere on the filesystem before they can be executed ([see this JRuby issue](#)). To work around this issue, you can either mount the `/tmp` directory without `noexec`, or you can choose a different directory to use as the temporary directory for the Puppet Server process.

Either way, you'll need to set the permissions of the directory to `1777`. This allows the Puppet Server JRuby process to write a file to `/tmp` and then execute it. If permissions are set incorrectly, you'll get a massive stack trace without much useful information in it.

To use a different temporary directory, you can set the following JVM property:

```
-Djava.io.tmpdir=/some/other/temporary/directory
```

When Puppet Server is installed from packages, add this property to the `JAVA_ARGS` and `JAVA_ARGS_CLI` variables defined in either `/etc/sysconfig/puppetserver` or `/etc/default/puppetserver`, depending on your distribution. Invocations of the `gem`, `ruby`, and `irb` subcommands use the updated `JAVA_ARGS_CLI` on their next invocation. The service will need to be restarted in order to re-read the `JAVA_ARGS` variable.

Puppet Server Fails to Connect to Load-Balanced Servers with Different SSL Certificates

SERVER-207: Intermittent SSL connection failures have been seen when Puppet Server tries to make SSL requests to servers via the same virtual ip address. This has been seen when the servers present different certificates during the SSL handshake.

What's new since Puppet 6?

These are the major new features, enhancements, deprecations, and removals since the Puppet 6 release.

Note that this list is not intended to be exhaustive. For all changes, see the [release notes](#).

Get familiar with the latest hardware requirements, supported operating systems and browsers, and network configuration details in [System requirements](#).

Factor 4

Factor 4 introduces new features, including granular blocking and caching of all types of facts, user defined fact groups, fact hierarchies using the `dot` notation and profiling using the `--timing` option. Factor 4 is written in Ruby, instead of C++. It is API compatible with Factor 3, but there may be some inconsistencies. Puppet 7 drops support for Factor 3. For a full list of new features and enhancements, see the [Factor 4 release notes](#).

Ruby 2.7

Puppet 7 agents have upgraded to Ruby 2.7 and dropped support for Ruby 2.3 and 2.4. After upgrading to Puppet 7, use the `puppet_gem` provider to ensure all your gems are installed.

Postgres 11+

PuppetDB now requires Postgres 11+, which allows us to write faster migrations and take advantage of newer features like logical partitioning.

Environment caching

The Puppet 7 `environment_timeout` behaves differently for values that are not 0 or unlimited. Puppet Server keeps your most actively used environments cached, but allows testing environments to fall out of the cache and reduce memory usage.

SHA256

Puppet 7 defaults to using SHA256 for all digest operations. MD5 is still be available for non-FIPS platforms, but you must opt into it using the `checksum` parameter for a `file` resource.



CAUTION: To avoid breaking changes when upgrading, either disable remote filebuckets or make sure the agent has the same digest algorithm as server by changing the `digest_algorithm` setting on the agent to `sha256`.

CA directory

We have made changes to prevent you from accidentally deleting your CA directory. This change is backwards compatible.

Puppet language enhancements

The Puppet 7 compiler raises syntax errors if it encounters application orchestration language keywords. These keywords remain reserved for future use.

Platform end-of-life

We have dropped agent support for the following platforms: EL5, Debian 8, SLES 11, Ubuntu 14.04, and Windows 2008/2008R2. Puppet Server and PuppetDB platforms have not changed.

Removals

We've removed the following deprecated functionality:

- Legacy authorization — replaced by Puppet Server's `auth.conf` in Puppet 5.
- Legacy routes — Puppet 3 agents are no longer be able to communicate with these.
- Puppet `key`, `cert` and `status` commands.

For a full list of removals and deprecations, see the [Puppet 7 release notes](#).

Documentation terminology changes

Documentation for this release replaces the term `master` with `primary server`. This change is part of a company-wide effort to remove harmful terminology from our products. For the immediate future, you'll continue to encounter `master` within the product, for example in parameters, commands, and preconfigured node groups. Where documentation references these codified product elements, we've left the term as-is. As a result of this update, if you've bookmarked or linked to specific sections of a docs page that include `master` in the URL, you'll need to update your link.

Installing and configuring

- [Installing and upgrading](#) on page 39

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

- [Configuring Puppet settings](#) on page 56

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

Installing and upgrading

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

- [System requirements](#) on page 39

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

- [Installing Puppet](#) on page 41

To get started using Puppet, you must first complete the initial installation and setup process.

- [Installing and configuring agents](#) on page 43

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

- [Manually verify packages](#) on page 50

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

- [Managing Platform versions](#) on page 53

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

- [Upgrading](#) on page 53

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

System requirements

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

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

Hardware requirements

The primary server is fairly resource intensive, and must be installed on a robust, dedicated server. The agent service has few system requirements and can run on nearly anything.

The Puppet agent has been run and tested on a variety of hardware specifications. These are the weakest hardware specifications we have successfully tested. Both were tested using Intel Xeon processors.

CPUs	GHz	GiB memory	OS
1	2.4	0.5	Amazon Linux 2 AMI
1	2.5	1	Windows Server 2019

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

These minimum hardware requirements are based on internal testing.

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

Supported agent platforms

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

Packaged platforms

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

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

Operating system	Tested versions	Untested versions
Debian	9, 10	
Fedora	30, 31, 32	
macOS	10.14 Mojave (see note below), 10.15 Catalina	
Microsoft Windows	10 Enterprise	8, 10
Microsoft Windows Server	2012R2, 2016, 2019	2012
Red Hat Enterprise Linux, including:	6, 7, 8	
• Amazon Linux v1 (using RHEL 6 packages)		
• Amazon Linux v2 (using RHEL 7 packages)		
SUSE Linux Enterprise Server	12, 15	
Ubuntu	16.04, 18.04, 20.04, 20.04 AARCH	

Note: On macOS 10.14 Mojave and later, you must grant Puppet Full Disk Access to be able to manage users and groups. To give Puppet access, go to **System Preferences > Security & Privacy > Privacy > Full Disk Access**, and add the path to the Puppet executable. For detailed steps, see [Add full disk access for Puppet on macOS 10.14 and newer](#) on page 47. Alternatively, set up automatic access using Privacy Preferences Control Profiles and a Mobile Device Management Server.

Dependencies

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

- Ruby 2.5.x
- Facter 2.0 or later
- Optional gems such `hiera-eyaml`, `hocon`, `msgpack`, `ruby-shadow`, etc.

Timekeeping and name resolution

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

Timekeeping

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

Name resolution

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

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

Firewall configuration

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

Installing Puppet

To get started using Puppet, you must first complete the initial installation and setup process.

Puppet is distributed in several packages. These include `puppetserver`, `puppet-agent` and `puppetdb`. Puppet Server controls the configuration information for one or more managed agent nodes. PuppetDB is where the data generated by Puppet is stored.

This guide walks you through the following steps in installing Puppet:

- Enabling the Puppet platform repository
- Installing Puppet Server
- Installing Puppet agent
- Installing PuppetDB (optional)

You install each of these components separately, operating on a single node. From here, you can scale up to the large installation as your infrastructure grows, or customize configuration as needed.

Note: The `puppetserver` component of the Puppet platform is available only for Linux. The `puppet-agent` component is available independently for over 30 platforms and architectures, including Windows and macOS. For more information on Puppet's packages, see [Puppet platform lifecycle](#).

1. Enable the Puppet platform repository

Enabling the Puppet platform repository makes the components needed for installation available on your system. The process for enabling the repository varies based on your package management system.

Before you begin

Identify the URL of the package you want to enable based on your operating system and version. *nix platform packages are located in a Puppet.com repository corresponding to the package management system.

Package management system	URL naming convention	URL example
Yum (Used with Red Hat operating systems such as Red Hat Enterprise Linux (RHEL) and SUSE Linux Enterprise Server.)	<code>https://yum.puppet.com/ <PLATFORM_NAME>-release- <OS ABBREVIATION>-<OS VERSION>.noarch.rpm</code>	<code>https://yum.puppet.com/ puppet7-release- el-8.noarch.rpm</code>
Apt (Used with Debian and Ubuntu.)	<code>https://apt.puppet.com/ <PLATFORM_VERSION>- release-<VERSION CODE NAME>.deb</code> Tip: For Ubuntu releases, the code name is the adjective, not the animal.	<code>https://apt.puppet.com/ puppet7-release-focal.deb</code>

Enable the Puppet platform on Yum

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

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

For example, to enable the Enterprise Linux 8 repository:

```
sudo rpm -Uvh https://yum.puppet.com/puppet7-release-el-8.noarch.rpm
```

Enable the Puppet platform on Apt

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

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

For example, to enable the Ubuntu 20.04 focal repository:

```
wget https://apt.puppet.com/puppet7-release-focal.deb  
sudo dpkg -i puppet7-release-focal.deb
```

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

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

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

If you need to manually verify packages, see [Verify packages](#).

2. Install Puppet Server

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

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

In this section, you will install the `puppetserver` package and start the service.

Follow the steps in [install Puppet Server](#)

3. Install Puppet agent

Puppet agents translates code into commands and then executes it on the systems you specify.

In this section, you will install agents on your chosen operating system, configure them, and sign their certificates. Follow the steps in [install agents](#).

4. Install PuppetDB (optional)

All of the data generated by Puppet is stored in Puppet DB.

You can optionally install PuppetDB to enable extra features, including enhanced queries and reports about your infrastructure. In this section, you will assign PuppetDB module's classes to your servers. Follow the steps in [install PuppetDB](#).

Installing and configuring agents

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

After you install an agent, you must complete the steps in [Configure agents](#).

Install agents

Install *nix agents

You can install *nix agents using an install script.

Before you begin

[Enable the Puppet platform repository](#).

1. Install the agent using the command appropriate to your environment.

- Yum:

```
sudo yum install puppet-agent
```

- Apt:

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

Note: The Puppet repository for the APT package management system is: <http://apt.puppet.com/>

- Zypper:

```
sudo zypper install puppet-agent
```

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

Install Windows agents

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

Install Windows agents with the .msi package

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

Before you begin

[Download](#) the .msi package.

Install Windows agents with the installer

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

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

Install Windows agents using `msiexec` from the command line

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

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

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

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

MSI properties

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

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

Property	Definition	Setting in <code>pe.conf</code>	Default
<code>INSTALLDIR</code>	Location to install Puppet and its dependencies.	n/a	<ul style="list-style-type: none"> 32-bit — C:\Program Files\Puppet Labs\Puppet 64-bit — C:\Program Files\Puppet Labs\Puppet
<code>PUPPET_SERVER</code>	Hostname where the primary server can be reached.	<code>server</code>	<code>puppet</code>
<code>PUPPET_CA_SERVER</code>	Hostname where the CA server can be reached, if you're using multiple servers and only one of them is acting as the CA.	<code>ca_server</code>	Value of <code>PUPPET_SERVER</code>
<code>PUPPET_AGENT_CERTNAME</code>	Node's certificate name, and the name it uses when requesting catalogs. For best compatibility, limit the value of <code>certname</code> to lowercase letters, numbers, periods, underscores, and dashes.	<code>certname</code>	Value of <code>facter.fqdn</code>

Property	Definition	Setting in <code>pe.conf</code>	Default
PUPPET_AGENT_ENVIRONMENT	<p>Node's environment.</p> <p>Note: If a value for the environment variable already exists in <code>puppet.conf</code>, specifying it during installation does not override that value.</p>	environment	production
PUPPET_AGENT_STARTUP	<p>Whether and how the agent service is allowed to run. Allowed values are:</p> <ul style="list-style-type: none"> Automatic — Agent starts up when Windows starts and remains running in the background. Manual — Agent can be started in the services console or with <code>net start</code> on the command line. Disabled — Agent is installed but disabled. You must change its startup type in the services console before you can start the service. 	n/a	Automatic

Property	Definition	Setting in <code>pe.conf</code>	Default
PUPPET_AGENT_ACCOUNT_USER	<p>Windows user account the agent service uses. This property is useful if the agent needs to access files on UNC shares, because the default LocalService account can't access these network resources.</p> <p>The user account must already exist, and can be either a local or domain user. The installer allows domain users even if they have not accessed the machine before. The installer grants Logon as Service to the user, and if the user isn't already a local administrator, the installer adds it to the Administrators group.</p> <p>This property must be combined with PUPPET_AGENT_ACCOUNT_PASSWORD and PUPPET_AGENT_ACCOUNT_DOMAIN.</p>	n/a	LocalSystem
PUPPET_AGENT_ACCOUNT_PASSWORD	Password for the agent's user account.	n/a	No Value
PUPPET_AGENT_ACCOUNT_DOMAIN	Domain of the agent's user account.	n/a	.
REINSTALLMODE	<p>A default MSI property used to control the behavior of file copies during installation.</p> <p>Important: If you need to downgrade agents, use REINSTALLMODE=amus when calling <code>msiexec.exe</code> at the command line to prevent removing files that the application needs.</p>	n/a	<p>amus as of puppet-agent 1.10.10 and puppet-agent 5.3.4</p> <p>omus in prior releases</p>

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

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

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

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

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

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

Upgrading to 64-bit

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

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

Downgrading to 32-bit

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

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

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

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

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

Install macOS agents

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

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

Add full disk access for Puppet on macOS 10.14 and newer

Beginning with macOS 10.14, you must add Puppet to the full disk access list, or allowlist, in order to run Puppet with full permissions and for it to properly manage resources like `user` and `group` on your system.

Complete these steps before attempting to install macOS agents.

1. Run the following command to remove the `.sh` extension from the `wrapper.sh` file:

```
mv /opt/puppetlabs/puppet/bin/wrapper.sh /opt/puppetlabs/puppet/bin/
wrapper
```

2. Run the following commands to relink `factor`, `hiera`, and `puppet` with the newly renamed file:

```
ln -sf /opt/puppetlabs/puppet/bin/wrapper /opt/puppetlabs/bin/facter
```

```
ln -sf /opt/puppetlabs/puppet/bin/wrapper /opt/puppetlabs/bin/hiera
```

```
ln -sf /opt/puppetlabs/puppet/bin/wrapper /opt/puppetlabs/bin/puppet
```

3. In your Mac **Preferences**, click **Security & Privacy**, select the **Privacy** tab, and click **Full Disk Access** in the left column.
4. Click the lock icon, enter your password, and click **Unlock**.
5. Click the + button, then type the # (**Command**) + **Shift** + **G** shortcut key.
6. Enter `/opt/puppetlabs/bin`, then click **Go**.
7. Click on the **puppet** file, then click **Open**.

Install macOS agents from Finder

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

Before you begin

[Download](#) the appropriate agent tarball.

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

You must include the primary server hostname and the agent certname.

Install macOS agents from the command line

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

Before you begin

[Download](#) the appropriate agent tarball.

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

A line appears ending with `/Volumes/puppet-agent-VERSION`. This directory location is the mount point for the virtual volume created from the disk image.

3. Change to the directory indicated as the mount point in the previous step, for example: `cd /Volumes/puppet-agent-VERSION`
4. Install the agent package: `sudo installer -pkg puppet-agent-installer.pkg -target /`
5. Verify the installation: `/opt/puppetlabs/bin/puppet --version`

Install macOS agents with Homebrew

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

Before you begin

Install [Homebrew](#).

Install the latest version of the Puppet agent:

```
brew cask install puppetlabs/puppet/puppet-agent
```

Configure agents

Once you have installed your agents, you must complete the following three configuration steps.

1. Configure your PATH to access Puppet commands

Puppet's command line interface (CLI) consists of a [single Puppet command with many subcommands](#), for example `puppet --help`.

Puppet commands are located in the bin directory — `/opt/puppetlabs/bin/` on *nix and `C:\Program Files\Puppet Labs\puppet\bin` on Windows. The bin directory is not in your PATH environment variable by default. To have access to the Puppet commands, you must add the bin directory to your PATH.

Choose from the following options.

Linux: source a script for puppet-agent to install

If you are on Linux, you can source a script that `puppet-agent` installs. Run the following command:

```
source /etc/profile.d/puppet-agent.sh
```

*nix: Add the Puppet labs bin directory to your PATH

To add the bin directory to your PATH on *nix, run:

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

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

Windows: Add the Puppet labs bin directory to your PATH

To run Puppet commands on Windows, start a command prompt with administrative privileges. You can do so by right-clicking the Start Command Prompts with Puppet program and clicking **Run** as administrator. Click **Yes** if the system asks for UAC confirmation.

The Puppet agent `.msi` adds the Puppet bin directory to the system path automatically. If you are not using the Start Command Prompts, you may need to manually add the bin directory to your PATH using one of the following commands:

For `cmd.exe`, run:

```
set PATH=%PATH%; "C:\Program Files\Puppet Labs\Puppet\bin"
```

For PowerShell, run:

```
$env:PATH += ";C:\Program Files\Puppet Labs\Puppet\bin"
```

2. Configure the server setting

The `server` setting, which allows you to connect the agent to the primary Puppet server, is the only mandatory setting.

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

To configure the server setting, choose from one of the following options:

- On the agent node, run:

```
puppet config set server puppetserver.example.com --section main
```

- Manually edit `/etc/puppetlabs/puppet/puppet.conf` or `C:\ProgramData\PuppetLabs\puppet\etc\puppet.conf`.

Note that the location on Windows depends on whether you are running with administrative privileges. If you are not, it will be in home directory, not system location.

This command adds the setting `server = puppetserver.example.com` to the `[main]` section of `puppet.conf`.

Note that there are other optional settings, for example, `serverport`, `ca_server`, `ca_port`, `report_server`, `report_port`, which you might need for more complicated Puppet deployments, such as when using a CA server and multiple compilers.

3. Connect the agent to the primary server and sign the certificate

Once you had added the `server`, you must connect the Puppet agent to the primary server so that it will check in at regular intervals to report its state, retrieve its catalog, and update its configuration if needed.

1. To connect the agent to the primary server, run:

```
puppet ssl bootstrap
```

Note: For Puppet 5 agents, run `puppet agent --test` instead.

You will see a message that looks like:

```
Info: Creating a new RSA SSL key for <agent node>
```

2. On the primary server node, sign the certificate:

```
sudo puppetserver ca sign --certname <name>
```

3. On the agent node, run the agent again:

```
puppet ssl bootstrap
```

Manually verify packages

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

Tip:

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

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

- [Verify a source tarball or gem](#) on page 50

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

- [Verify an RPM package](#) on page 51

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

- [Verify a macOS puppet-agent package](#) on page 52

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

Verify a source tarball or gem

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

1. Import the public key: `gpg --keyserver pgp.mit.edu --recv-key 4528B6CD9E61EF26`

The key is also available via [HTTP](http://pgp.mit.edu/pks/lookup?search=4528B6CD9E61EF26&server=pgp.mit.edu).

Tip: If this is your first time running the gpg tool, it might fail to import the key after creating its configuration file and keyring. You can run the command a second time to import the key into your newly created keyring. If it continues to fail, try `gpg --keyserver hkps://keyserver.ubuntu.com:11371 --recv-key 4528B6CD9E61EF26`

The gpg tool imports the key:

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

2. Verify the fingerprint: `gpg --list-key --fingerprint 4528B6CD9E61EF26`

The fingerprint of the Puppet release signing key is D6811ED3ADEEB8441AF5AA8F4528B6CD9E61EF26. Ensure the fingerprint listed matches this value.

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

The output confirms that the signature matches:

```
gpg: Signature made Mon 09 Nov 2020 12:19:14 PM PST using RSA key ID
9E61EF26
gpg: Good signature from "Puppet, Inc. Release Key (Puppet, Inc. Release
Key) <release@puppet.com>"
```

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

Verify an RPM package

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

1. Import the public key: `gpg --keyserver pgp.mit.edu --recv-key 4528B6CD9E61EF26`

The key is also available via [HTTP](http://pgp.mit.edu/pks/lookup?search=4528B6CD9E61EF26&server=pgp.mit.edu).

Tip: If this is your first time running the gpg tool, it might fail to import the key after creating its configuration file and keyring. You can run the command a second time to import the key into your newly created keyring. If it continues to fail, try `gpg --keyserver hkps://keyserver.ubuntu.com:11371 --recv-key 4528B6CD9E61EF26`

The gpg tool imports the key:

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

2. Verify the fingerprint: `gpg --list-key --fingerprint 4528B6CD9E61EF26`

The fingerprint of the Puppet release signing key is D6811ED3ADEEB8441AF5AA8F4528B6CD9E61EF26. Ensure the fingerprint listed matches this value.

3. Retrieve the [Puppet public key](#) and place it in a file on your node.

4. Use the RPM tool to import the public key, replacing <PUBLIC KEY FILE> with the path to the file containing the public key: `sudo rpm --import <PUBLIC KEY FILE>`

The RPM tool doesn't output anything if the command is successful.

5. Use the RPM tool to check the signature of a downloaded RPM package: `sudo rpm -vK <RPM_FILE_NAME>`

The embedded signature is verified and displays OK:

```
puppet-agent-1.5.1-1.el6.x86_64.rpm:
Header V4 RSA/SHA512 Signature, key ID EF8D349F: OK
Header SHA1 digest: OK (95b492a1fff452d029aaeb59598f1c78dbfee0c5)
V4 RSA/SHA512 Signature, key ID EF8D349F: OK
MD5 digest: OK (4878909ccdd0af24fa9909790dd63a12)
```

Verify a macOS puppet-agent package

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

Use one of these methods to verify the package signature:

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

```
/usr/bin/hdiutil attach puppet-agent-<AGENT-VERSION>-1.osx10.15.dmg
...
pkgutil --check-signature /Volumes/puppet-agent-<AGENT-VERSION>-1.osx10.15/puppet-agent-<AGENT-VERSION>-1-installer.pkg
```

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

```
Package "puppet-agent-<AGENT-VERSION>-1-installer.pkg":
Status: signed by a developer certificate issued by Apple for
distribution
Certificate Chain:
  1. Developer ID Installer: PUPPET LABS, INC. (VKGLGN2B6Y)
     SHA256 Fingerprint:
02      F9 6D CA EF 1B D8 FF 30 1D 25 67 54 90 CF 7F C3 BF 39 91 50 A6
        65 FA B2 19 4B 1E 2A B6 D1 9E
-----
  2. Developer ID Certification Authority
     SHA256 Fingerprint:
03      7A FC 9D 01 A6 2F 03 A2 DE 96 37 93 6D 4A FE 68 09 0D 2D E1 8D
        F2 9C 88 CF B0 B1 BA 63 58 7F
-----
  3. Apple Root CA
     SHA256 Fingerprint:
2C      B0 B1 73 0E CB C7 FF 45 05 14 2C 49 F1 29 5E 6E DA 6B CA ED 7E
        68 C5 BE 91 B5 A1 10 01 F0 24
```

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

The installer displays details about the package's certificate.

Managing Platform versions

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

For example, if you're using the `puppetlabs/puppet_agent` module to manage the installed `puppet-agent` package, use this resource to pin it to version 7.0.0:

```
class { '::puppet_agent':
  collection    => 'puppet7',
  package_version => '7.0.0',
}
```

To upgrade to newer versions within the `puppet7` collection, update the `package_version`. If you're upgrading from an earlier collection, simply update the `collection` and `package_version`.

Upgrading

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

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

Important: These instructions cover upgrades in the versions 5, 6 and 7 series. For instructions on upgrading from version 3.8.x, see previous versions of the documentation.

Upgrade Puppet Server

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

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

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

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

Yum:

```
yum update puppetserver
```

Apt:

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

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

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

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

Upgrade agents

Regularly upgrade agents to keep your systems running smoothly.

Before you begin

Upgrade Puppet Server.

Upgrade agents using the `puppet_agent` module

Upgrade your Puppet agents using the `puppetlabs/puppet_agent` module. The `puppet_agent` module supports upgrading open source Puppet agents on *nix, Windows, and macOS

Before you begin

Install the [puppetlabs/puppet_agent](#) module. Read about installing modules in the following docs:

- [Installing and managing modules from the command line](#) on page 599
- [Install modules on nodes without internet](#) on page 602



CAUTION: Puppet 7 changed the default `digest_algorithm` setting to `sha256`. To avoid breaking changes when upgrading, either disable remote filebuckets or make sure the agent has the same digest algorithm as server by changing the `digest_algorithm` setting on the agent to `sha256`.

To upgrade on a Puppet Server node:

1. Add the `puppet_agent` class to agents you want to upgrade.
2. Specify the desired `puppet_agent` package version and any other desired parameters described in the Forge in your `site.pp` file.

Note: The `collection` parameter is required in Open Source Puppet.

For example:

```
# site.pp
node default {
  class { puppet_agent:
    package_version => '7.0.0',
    collection      => 'puppet7'
  }
}
```

This upgrades all of your Puppet Server managed nodes on the next agent run.

To upgrade on a node that is **not** managed by Puppet Server, you must declare the class in your manifest.

For example, an upgrade manifest for a non-standard source looks like:

```
# upgrade_file.pp
class { puppet_agent:
  package_version => '7.0.0.224.gaf2034af',
  yum_source      => 'http://nightlies.puppet.com/yum',
  collection      => 'puppet7-nightly'
}
```

You then need to apply the manifest: `puppet apply upgrade_file.pp`

Upgrade *nix agents

We recommend using the [puppetlabs/puppet_agent](#) module when upgrading between major versions of Puppet agent. To upgrade *nix without the module, you can use the system's package manager.

Before upgrading to another major version, ensure that you have installed the appropriate release package for your platform. You can find release packages for supported platforms at yum.puppet.com and apt.puppet.com. We also provide nightly release packages at nightlies.puppet.com.

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

Yum (EL/Fedora):

```
yum update puppet-agent
```

Zypper (SLES)

```
zypper up puppet-agent
```

Apt (Debian/Ubuntu):

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

Upgrade Windows agents

To upgrade Windows agents without the [puppetlabs/puppet_agent](#) module, reinstall the agent using the installation instructions. You don't need to uninstall the agent before reinstalling unless you're upgrading from 32-bit Puppet to the 64-bit version.

Upgrade macOS agents

To upgrade macOS agents without the [puppetlabs/puppet_agent](#) module, use the `puppet resource` command.

Before you begin

[Download](#) the appropriate agent DMG.

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

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

Upgrade PuppetDB

Upgrade PuppetDB to get the newest features available.

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

Yum:

```
yum update puppetdb
```

Apt:

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

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

Yum:

```
yum update puppetdb-termini
```

Apt:

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

Configuring Puppet settings

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

- [Puppet settings](#) on page 56

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

- [Key configuration settings](#) on page 59

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

- [Puppet's configuration files](#) on page 62

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

- [Adding file server mount points](#) on page 88

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

- [Checking the values of settings](#) on page 90

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

- [Editing settings on the command line](#) on page 92

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

- [Configuration settings reference](#)
- [Differing behavior in puppet.conf](#) on page 93

Puppet settings

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

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

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

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

Puppet Server configuration

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

For more information, see [Puppet Server: Configuration](#).

Settings are loaded on startup

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

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

Settings on the command line

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

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

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

Basic settings

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

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

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

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

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

Boolean settings

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

This means:

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

Default values

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

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

Configuring locale settings

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

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

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

Checking your locale settings on *nix and macOS

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

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

```
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_ALL=
```

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

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

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

Checking your locale settings on Windows

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

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

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

Setting your locale on *nix with an environment variable

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

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

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

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

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

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

Setting your locale for the Puppet agent service on macOS

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

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

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

Setting your locale on Windows

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

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

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

Disabling internationalized strings

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

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

Key configuration settings

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

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

For a full list of Puppet settings, see the [configuration reference](#).

Settings for agents (all nodes)

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

Basics

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

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

Run behavior

These settings affect the way Puppet applies catalogs:

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

Service behavior

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

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

Useful when running agent from cron

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

For more information on these settings, see the [configuration reference](#).

Settings for primary servers

Many of these settings are also important for standalone Puppet apply nodes, because they act as their own primary server. These settings go in the `[server]` section, unless you're using Puppet apply in production, in which case put them in the `[main]` section instead.

Basics

- `dns_alt_names` — A list of hostnames the server is allowed to use when acting as a primary server. The hostname your agents use in their `server` setting must be included in either this setting or the primary server's `certname` setting. Note that this setting is only used when initially generating the primary server's certificate — if you need to change the DNS names, you must:
 1. Turn off the Puppet Server service (or your Rack server).
 2. Run: `sudo puppetserver ca clean <SERVER'S CERTNAME>`
 3. Run: `sudo puppetserver ca generate <SERVER'S CERTNAME> --dns-alt-names <ALT NAME 1>,<ALT NAME 2>,...`

4. Re-start the Puppet Server service.

- `environment_timeout` — For better performance, you can set this to `unlimited` and make refreshing the primary server a part of your standard code deployment process.
- `environmentpath` — Controls where Puppet finds directory environments. For more information on environments, see [Creating environments](#).
- `basemodulepath` — A list of directories containing Puppet modules that can be used in all environments. See [modulepath](#) for details.
- `reports` — Which report handlers to use. For a list of available report handlers, see [the report reference](#). You can also [write your own report handlers](#). Note that the report handlers might require settings of their own.
- `digest_algorithm` — To accept requests from older agents when using a remote filebucket, Puppet Server needs to specify `digest_algorithm=md5`.

Puppet Server related settings

Puppet Server has its own configuration files; consequently, there are [several settings in puppet.conf that Puppet Server ignores](#).

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

Extensions

These features configure add-ons and optional features:

- `node_terminus` and `external_nodes` — The ENC settings. If you're using an [ENC](#), set these to `exec` and the path to your ENC script, respectively.
- `storeconfigs` and `storeconfigs_backend` — Used for setting up PuppetDB. See [the PuppetDB docs for details](#).
- `catalog_terminus` — This can enable the optional static compiler. If you have lots of file resources in your manifests, the static compiler lets you sacrifice some extra CPU work on your primary server to gain faster configuration and reduced HTTPS traffic on your agents. See the [indirection reference](#) for details.

CA settings

- `ca_ttl` — How long newly signed certificates are valid. Deprecated.
- `autosign` — Whether and how to autosign certificates. See [Autosigning](#) for detailed information.

For more information on these settings, see the [configuration reference](#).

Puppet's configuration files

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

- [puppet.conf: The main config file](#) on page 62

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

- [environment.conf: Per-environment settings](#) on page 65

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

- [fileserver.conf: Custom fileserver mount points](#) on page 67

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

- [puppetdb.conf: PuppetDB server locations](#) on page 67

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

- [hiera.yaml: Data lookup configuration](#)

- [autosign.conf: Basic certificate autosigning](#) on page 67

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

- [csr_attributes.yaml: Certificate extensions](#) on page 68

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

- [custom_trusted_oid_mapping.yaml: Short names for cert extension OIDs](#) on page 70

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

- [device.conf: Network hardware access](#) on page 71

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

- [routes.yaml: Advanced plugin routing](#) on page 72

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

puppet.conf: The main config file

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

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

For a complete list of Puppet's settings, see the [configuration reference](#).

Location

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

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

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

The location of the `confdir` depends on your operating system. See the [confdir documentation](#) for details.

Examples

Example agent config:

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

Example server config:

```
[main]
certname = puppetserver01.example.com
server = puppet
runinterval = 1h
strict_variables = true

[server]
dns_alt_names =
  primaryserver01,primaryserver01.example.com,puppet,puppet.example.com
reports = puppetdb
storeconfigs_backend = puppetdb
storeconfigs = true
```

Format

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

The file can also include comment lines at any point.

Config sections

```
[main]
  certname = primaryserver01.example.com
```

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

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

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

Puppet uses four config sections:

- `main` is the global section used by all commands and services. It can be overridden by the other sections.
- `server` is used by the primary Puppet server service and the Puppet Server `ca` command.

Important: Be sure to apply settings only in `main` unless there is a specific case where you have to override a setting for the `server` run mode. For example, when Puppet Server is configured to use an external node classifier, you must add [these settings](#) to the `server` section. If those settings are added to `main`, then the agent tries and fails to run the server-only script `/usr/local/bin/puppet_node_classifier` during its run.

- `agent` is used by the Puppet agent service.
- `user` is used by the Puppet apply command, as well as many of the less common [Puppet subcommands](#).

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

Note: Puppet Server ignores some config settings. It honors almost all settings in `puppet.conf` and picks them up automatically. However, some [Puppet Server settings](#) differ from a Ruby primary server's `puppet.conf` settings.

Comment lines

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

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

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

Setting lines

```
certname = primaryserver01.example.com
```

A setting line consists of:

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

Special types of values for settings

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

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

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

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

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

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

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

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

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

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

Interpolating variables in settings

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

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

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

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

environment.conf: Per-environment settings

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

Location

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

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

Example

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

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

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

Format

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

Relative paths in values

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

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

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

Interpolation in values

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

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

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

Allowed settings

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

`modulepath`

The list of directories Puppet loads modules from.

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

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

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

`manifest`

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

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

`config_version`

A script Puppet can run to determine the configuration version.

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

You can specify an executable script that determines an environment's config version by setting `config_version` in its `environment.conf` file. Puppet runs this script when compiling a catalog for a node in the environment, and use its output as the config version.

Note: If you're using a system binary like `git rev-parse`, make sure to specify the absolute path to it. If `config_version` is set to a relative path, Puppet looks for the binary in the environment, not in the system's `PATH`.

If this setting isn't set, the config version is the time at which the catalog was compiled (as the number of seconds since January 1, 1970). A directory environment never uses the global `config_version` from `puppet.conf`.

`environment_timeout`

How long the primary server caches the data it loads from an environment. If present, this overrides the value of `environment_timeout` from [puppet.conf](#). Unless you have a specific reason, we recommend only setting `environment_timeout` globally, in `puppet.conf`. We also don't recommend using any value other than 0 or unlimited.

For more information about configuring the environment timeout, [see the timeout section of the Creating Environments page](#).

fileserver.conf: Custom fileserver mount points

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

When to use fileserver.conf

This file is necessary only if you are [creating custom mount points](#).

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

Location

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

The location of the `confdir` depends on your operating system. See the [confdir documentation](#) for details.

Example

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

This `fileserver.conf` file creates a new mount point named `extra_files`. Authorization to `extra_files` is controlled by Puppet Server. See [creating custom mount points](#) for more information.



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

Format

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

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

puppetdb.conf: PuppetDB server locations

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

This configuration file is documented in the PuppetDB docs. See [Configuring a Puppet/PuppetDB connection](#) for details.

autosign.conf: Basic certificate autosigning

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



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

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

For more information, see the documentation about [certificate autosigning](#).

Location

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

The default `confdir` path depends on your operating system. See the [confdir documentation](#) for more information.

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

Format

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

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

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

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

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

`csr_attributes.yaml`: Certificate extensions

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

The `csr_attributes.yaml` file can set:

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

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

For information about using this file, see [CSR attributes and certificate extensions](#).

Location

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

The location of the `confdir` depends on your operating system. See the [confdir documentation](#) for details.

Example

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

Format

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

- `custom_attributes`
- `extension_requests`

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

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

Allowed OIDs for custom attributes

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

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

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

Allowed OIDs for extension requests

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

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

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

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

The “ppRegCertExt” OID range contains the following OIDs.

Numeric ID	Short name	Descriptive name
1.3.6.1.4.1.34380.1.1.1	<code>pp_uuid</code>	Puppet node UUID
1.3.6.1.4.1.34380.1.1.2	<code>pp_instance_id</code>	Puppet node instance ID
1.3.6.1.4.1.34380.1.1.3	<code>pp_image_name</code>	Puppet node image name
1.3.6.1.4.1.34380.1.1.4	<code>pp_preshared_key</code>	Puppet node preshared key
1.3.6.1.4.1.34380.1.1.5	<code>pp_cost_center</code>	Puppet node cost center name
1.3.6.1.4.1.34380.1.1.6	<code>pp_product</code>	Puppet node product name
1.3.6.1.4.1.34380.1.1.7	<code>pp_project</code>	Puppet node project name
1.3.6.1.4.1.34380.1.1.8	<code>pp_application</code>	Puppet node application name
1.3.6.1.4.1.34380.1.1.9	<code>pp_service</code>	Puppet node service name
1.3.6.1.4.1.34380.1.1.10	<code>pp_employee</code>	Puppet node employee name
1.3.6.1.4.1.34380.1.1.11	<code>pp_created_by</code>	Puppet node created_by tag

Numeric ID	Short name	Descriptive name
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

The “ppAuthCertExt” OID range contains the following OIDs:

1.3.6.1.4.1.34380.1.3.1	pp_authorization	Certificate extension authorization
1.3.6.1.4.1.34380.1.3.13	pp_auth_role	Puppet node role name for authorization. For PE internal use only.

custom_trusted_oid_mapping.yaml: Short names for cert extension OIDs

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

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

Certificate extensions

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

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

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

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

For more information, see [CSR attributes and certificate extensions](#), [Trusted facts](#), [The `csr_attributes.yaml` file](#).

Limitations of OID mapping

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

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

Location

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

The location of the `confdir` depends on your OS. See the [confdir documentation](#) for details.

Example

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

Format

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

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

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

device.conf: Network hardware access

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

For more information on Puppet device, see the [Puppet device documentation](#).

Location

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

The location of `confdir` depends on your operating system. See the [confdir documentation](#) for details.

Format

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

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

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

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

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

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

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

With:

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

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

routes.yaml: Advanced plugin routing

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

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

Location

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

The location of the `confdir` depends on your operating system. See the [confdir documentation](#) for details.

Example

```
---
primary server:
  facts:
    terminus: puppetdb
    cache: yaml
```

Format

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

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

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

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

Configuring Puppet Server

Configuring Puppet Server

Puppet Server automatically loads the settings in the `main` and `server` sections of the configuration file. If there are duplicates, it prefers the values in the `server` section. Puppet Server honors the following `puppet.conf` settings:

- `allow_duplicate_certs`
- `autosign`
- `cacert`

- `cacrl`
- `cakey`
- `ca_name`
- `capub`
- `ca_ttl`
- `certdir`
- `certname`
- `cert_inventory`
- `codedir` (PE only)
- `csrdir`
- `csr_attributes`
- `dns_alt_names`
- `hostcert`
- `hostcrl`
- `hostprivkey`
- `hostpubkey`
- `keylength`
- `localcacert`
- `manage_internal_file_permissions`
- `privatekeydir`
- `requestdir`
- `serial`
- `signedir`
- `ssl_client_header`
- `ssl_client_verify_header`
- `trusted_oid_mapping_file`

However, for some tasks, such as configuring the web server or an external Certificate Authority (CA), Puppet Server has separate configuration files and settings. These files and settings are described below. For more information about differences between Puppet Server and Ruby Puppet's use of `puppet.conf` settings, see [Differing behavior in puppet.conf](#) on page 93.

Configuration Files

Puppet Server's configuration files and settings (with the exception of the [logging config file](#)) are in the `conf.d` directory, located by default at `/etc/puppetlabs/puppetserver/conf.d`. These config files are in the HOCON format, which keeps the basic structure of JSON but is more readable. For more information, see the [HOCON documentation](#).

At startup, Puppet Server reads all the `.conf` files in the `conf.d` directory. You must restart Puppet Server for any changes to those files to take effect. The `conf.d` directory contains the following files and settings:

- [global.conf](#) on page 83
- [webserver.conf](#) on page 82
- [web-routes.conf](#) on page 82
- [puppetserver.conf](#) on page 75
- [auth.conf](#) on page 78
- [ca.conf](#) on page 83

The [product.conf](#) on page 83 file is optional and is not included by default. You can create that file in the `conf.d` directory in order to configure product-related settings, such as automatic update checking and analytics data collection.

Logging

Puppet Server's logging is routed through the JVM [Logback](#) library. The default Logback configuration file is at `/etc/puppetserver/logback.xml` or `/etc/puppetlabs/puppetserver/logback.xml`. You can edit this file to change the logging behavior, or specify a different Logback config file in [global.conf](#).

For more information on the `logback.xml` file, see [logback.xml](#) on page 84 and the [Logback documentation](#). For advanced logging configuration tips, such as configuring Logstash or outputting logs in JSON format, see [Advanced logging configuration](#) on page 85.

For some tips on advanced logging configuration, including information about configuring your system to write logs in a JSON format suitable for sending to logstash or other external logging systems, see the [Advanced logging configuration](#) on page 85 documentation.

HTTP Traffic

Puppet Server logs HTTP traffic in a format similar to Apache, and to a separate file than the main log file. By default, this is located at `/var/log/puppetlabs/puppetserver/puppetserver-access.log`.

By default, the following information is logged for each HTTP request:

- remote host
- remote log name
- remote user
- date of the logging event
- URL requested
- status code of the request
- response content length
- remote IP address
- local port
- elapsed time to serve the request, in milliseconds

The Logback configuration file is at `/etc/puppetlabs/puppetserver/request-logging.xml`. You can edit this file to change the logging behavior. Specify a different Logback configuration file in [webserver.conf](#) with the [access-log-config](#) setting. For more information on configuring the logged data, see [Logback Access Pattern Layout](#).

Authorization

To enable additional logging related to `auth.conf`, edit Puppet Server's `logback.xml` file. By default, only a single message is logged when a request is denied.

To enable a one-time logging of the parsed and transformed `auth.conf` file, add the following to Puppet Server's `logback.xml` file:

```
<logger name="puppetlabs.trapperkeeper.services.authorization.authorization-service" level="DEBUG"/>
```

To enable rule-by-rule logging for each request as it's checked for authorization, add the following to Puppet Server's `logback.xml` file:

```
<logger name="puppetlabs.trapperkeeper.authorization.rules" level="TRACE"/>
```

Service Bootstrapping

Puppet Server is built on top of our open-source Clojure application framework, [Trapperkeeper](#).

One of the features that Trapperkeeper provides is the ability to enable or disable individual services that an application provides. In Puppet Server, you can use this feature to enable or disable the CA service. The CA service is enabled by default, but if you're running a multi-server environment or using an external CA, you might want to disable the CA service on some nodes.

Starting in Puppet Server 2.5.0, the service bootstrap configuration files are in two locations:

- `/etc/puppetlabs/puppetserver/services.d/`: For services that users are expected to manually configure if necessary, such as CA-related settings.
- `/opt/puppetlabs/server/apps/puppetserver/config/services.d/`: For services users *shouldn't* need to configure.

Any files with a `.cfg` extension in either of these locations are combined to form the final set of services Puppet Server will use.

The CA-related configuration settings are set in `/etc/puppetlabs/puppetserver/services.d/ca.cfg`. If services added in future versions have user-configurable settings, the configuration files will also be in this directory. When upgrading Puppet Server 2.5.0 and newer with a package manager, it should not overwrite files already in this directory.

In the `ca.cfg` file, find and modify these lines as directed to enable or disable the service:

```
# To enable the CA service, leave the following line uncommented
puppetlabs.services.ca.certificate-authority-service/certificate-authority-
service
# To disable the CA service, comment out the above line and uncomment the
  line below
#puppetlabs.services.ca.certificate-authority-disabled-service/certificate-
authority-disabled-service
```

Adding Java JARs

Starting with Puppet Server 5.1.0, you can provide Java JARs to be loaded upon Puppet Server's startup.

When launched, Server automatically loads any JARs placed in `/opt/puppetlabs/server/data/puppetserver/jars` into the `classpath`. JARs placed here will not be modified or removed when upgrading Puppet Server.

puppetserver.conf

The `puppetserver.conf` file contains settings for the Puppet Server application. For an overview, see [Configuring Puppet Server](#) on page 72.

Settings

Note: Under most conditions, you won't change the default settings for `server-conf-dir` or `server-code-dir`. However, if you do, also change the equivalent Puppet settings (`confdir` or `codedir`) to ensure that commands like `puppetserver ca` and `puppet module` use the same directories as Puppet Server. You must also specify the non-default `confdir` when running commands, because that setting must be set before Puppet tries to find its config file.

- The `jruby-puppet` settings configure the interpreter.

Deprecation Note: Puppet Server 5.0 removed the `compat-version` setting, which is incompatible with JRuby 1.7.27, and the service won't start if `compat-version` is set. Puppet Server 6.0 uses JRuby 9.1 which supports Ruby 2.3.

- `ruby-load-path`: The location where Puppet Server expects to find Puppet, Facter, and other components.
- `gem-home`: The location where JRuby looks for gems. It is also used by the `puppetserver gem` command line tool. If nothing is specified, JRuby uses the Puppet default `/opt/puppetlabs/server/data/puppetserver/jruby-gems`.
- `gem-path`: The complete "GEM_PATH" for jruby. If set, it should include the `gem-home` directory, as well as any other directories that gems can be loaded from (including the vendored gems directory for gems that ship with puppetserver). The default value is `["/opt/puppetlabs/server/data/puppetserver/`

```
jruby-gems", "/opt/puppetlabs/server/data/puppetserver/vendored-jruby-gems", "/opt/puppetlabs/puppet/lib/ruby/vendor_gems"].
```

- `environment-vars`: Optional. A map of environment variables which are made visible to Ruby code running within JRuby, for example, via the Ruby ENV class.

By default, the only environment variables whose values are set into JRuby from the shell are HOME and PATH.

The default value for the GEM_HOME environment variable in JRuby is set from the value provided for the `jruby-puppet.gem-home` key.

Any variable set from the map for the `environment-vars` key overrides these defaults. Avoid overriding HOME, PATH, or GEM_HOME here because these values are already configurable via the shell or `jruby-puppet.gem-home`.

- `server-conf-dir`: Optional. The path to the Puppet [configuration directory](#). The default is `/etc/puppetlabs/puppet`.
- `server-code-dir`: Optional. The path to the Puppet [code directory](#). The default is `/etc/puppetlabs/code`.
- `server-var-dir`: Optional. The path to the Puppet [cache directory](#). The default is `/opt/puppetlabs/server/data/puppetserver`.
- `server-run-dir`: Optional. The path to the run directory, where the service's PID file is stored. The default is `/var/run/puppetlabs/puppetserver`.
- `server-log-dir`: Optional. The path to the log directory. If nothing is specified, it uses the Puppet default `/var/log/puppetlabs/puppetserver`.
- `max-active-instances`: Optional. The maximum number of JRuby instances allowed. The default is 'num-cpus - 1', with a minimum value of 1 and a maximum value of 4. In multithreaded mode, this controls the number of threads allowed to run concurrently through the single JRuby instance.
- `max-requests-per-instance`: Optional. The number of HTTP requests a given JRuby instance will handle in its lifetime. When a JRuby instance reaches this limit, it is flushed from memory and replaced with a fresh one. The default is 0, which disables automatic JRuby flushing.

JRuby flushing can be useful for working around buggy module code that would otherwise cause memory leaks, but it slightly reduces performance whenever a new JRuby instance reloads all of the Puppet Ruby code. If memory leaks from module code are not an issue in your deployment, the default value of 0 performs best.

- `multithreaded`: Optional, false by default. Configures Puppet Server to use a single JRuby instance to process requests that require a JRuby, processing a number of threads up to `max-active-instances` at a time. Reduces the memory footprint of the server by only requiring a single JRuby.
- `max-queued-requests`: Optional. The maximum number of requests that may be queued waiting to borrow a JRuby from the pool. When this limit is exceeded, a 503 "Service Unavailable" response will be returned for all new requests until the queue drops below the limit. If `max-retry-delay` is set to a positive value, then the 503 responses will include a `Retry-After` header indicating a random sleep time after which the client may retry the request. The default is 0, which disables the queue limit.
- `max-retry-delay`: Optional. Sets the upper limit for the random sleep set as a `Retry-After` header on 503 responses returned when `max-queued-requests` is enabled. A value of 0 will cause the `Retry-After` header to be omitted. Default is 1800 seconds which corresponds to the default run interval of the Puppet daemon.
- `borrow-timeout`: Optional. The timeout in milliseconds, when attempting to borrow an instance from the JRuby pool. The default is 1200000.
- `environment-class-cache-enabled`: Optional. Used to control whether the master service maintains a cache in conjunction with the use of the [Environment classes](#) on page 192.

If this setting is set to `true`, Puppet Server maintains the cache. It also returns an Etag header for each GET request to the API. For subsequent GET requests that use the prior Etag value in an `If-None-Match` header,

when the class information available for an environment has not changed, Puppet Server returns an HTTP 304 (Not Modified) response with no body.

If this setting is set to `false` or is not specified, Puppet Server doesn't maintain a cache, an Etag header is not returned for GET requests, and the If-None-Match header for an incoming request is ignored. It therefore parses the latest available code for an environment from disk on every incoming request.

For more information, see the [Environment classes](#) on page 192.

- `compile-mode`: The default value depends on JRuby versions, for 1.7 it is `off`, for 9k it is `jit`. Used to control JRuby's "CompileMode", which may improve performance. A value of `jit` enables JRuby's "just-in-time" compilation of Ruby code. A value of `force` causes JRuby to attempt to pre-compile all Ruby code.
- `profiling-mode`: Optional. Used to enable JRuby's profiler for service startup and set it to one of the supported modes. The default value is `off`, but it can be set to one of `api`, `flat`, `graph`, `html`, `json`, `off`, and `service`. See [ruby-prof](#) for details on what the various modes do.
- `profiler-output-file`: Optional. Used to set the output file to direct JRuby profiler output. Should be a fully qualified path writable by the service user. If not set will default to a random name inside the service working directory.
- The profiler settings configure profiling:
 - `enabled`: If this is set to `true`, Puppet Server enables profiling for the Puppet Ruby code. The default is `true`.
- The versioned-code settings configure commands required to use [static catalogs](#):
 - `code-id-command`: the path to an executable script that Puppet Server invokes to generate a `code_id`. When compiling a static catalog, Puppet Server uses the output of this script as the catalog's `code_id`. The `code_id` associates the catalog with the compile-time version of any [file resources](#) that has a `source` attribute with a `puppet:///URI` value.
 - `code-content-command` contains the path to an executable script that Puppet Server invokes when an agent makes a [Static file content](#) on page 200 API request for the contents of a [file resource](#) that has a `source` attribute with a `puppet:///URI` value.

Note: The Puppet Server process must be able to execute the `code-id-command` and `code-content-command` scripts, and the scripts must return valid content to standard output and an error code of 0. For more information, see the [static catalogs](#) and [Static file content](#) on page 200 documentation.

If you're using static catalogs, you **must** set and use **both** `code-id-command` and `code-content-command`. If only one of those settings are specified, Puppet Server fails to start. If neither setting is specified, Puppet Server defaults to generating catalogs without static features even when an agent requests a static catalog, which the agent will process as a normal catalog.

Examples

```
# Configuration for the JRuby interpreters.

jruby-puppet: {
  ruby-load-path: [/opt/puppetlabs/puppet/lib/ruby/vendor_ruby]
  gem-home: /opt/puppetlabs/server/data/puppetserver/jruby-gems
  gem-path: [/opt/puppetlabs/server/data/puppetserver/jruby-gems, /opt/
puppetlabs/server/data/puppetserver/vendored-jruby-gems]
  environment-vars: { "FOO" : ${FOO}
                      "LANG" : "de_DE.UTF-8" }
  server-conf-dir: /etc/puppetlabs/puppet
  server-code-dir: /etc/puppetlabs/code
  server-var-dir: /opt/puppetlabs/server/data/puppetserver
  server-run-dir: /var/run/puppetlabs/puppetserver
  server-log-dir: /var/log/puppetlabs/puppetserver
  max-active-instances: 1
  max-requests-per-instance: 0
}
```

```

# Settings related to HTTP client requests made by Puppet Server.
# These settings only apply to client connections using the
Puppet::Network::HttpPool
# classes. Client connections using net/http or net/https directly will not
be
# configured with these settings automatically.
http-client: {
  # A list of acceptable protocols for making HTTP requests
  #ssl-protocols: [TLSv1, TLSv1.1, TLSv1.2]

  # A list of acceptable cipher suites for making HTTP requests. For more
  info on available cipher suites, see:
  # http://docs.oracle.com/javase/7/docs/technotes/guides/security/
  SunProviders.html#SunJSSEProvider
  #cipher-suites: [TLS_RSA_WITH_AES_256_CBC_SHA256,
  #                TLS_RSA_WITH_AES_256_CBC_SHA,
  #                TLS_RSA_WITH_AES_128_CBC_SHA256,
  #                TLS_RSA_WITH_AES_128_CBC_SHA]

  # The amount of time, in milliseconds, that an outbound HTTP connection
  # will wait for data to be available before closing the socket. If not
  # defined, defaults to 20 minutes. If 0, the timeout is infinite and if
  # negative, the value is undefined by the application and governed by
the
  # system default behavior.
  #idle-timeout-milliseconds: 1200000

  # The amount of time, in milliseconds, that an outbound HTTP connection
will
  # wait to connect before giving up. Defaults to 2 minutes if not set. If
0,
  # the timeout is infinite and if negative, the value is undefined in the
  # application and governed by the system default behavior.
  #connect-timeout-milliseconds: 120000

  # Whether to enable http-client metrics; defaults to 'true'.
  #metrics-enabled: true
}

# Settings related to profiling the puppet Ruby code.
profiler: {
  enabled: true
}

# Settings related to static catalogs. These paths are examples. There are
no default
# scripts provided with Puppet Server, and no default path for the scripts.
To use static catalog features, you must set
# the paths and provide your own scripts.
versioned-code: {
  code-id-command: /opt/puppetlabs/server/apps/puppetserver/code-id-
command_script.sh
  code-content-command: /opt/puppetlabs/server/apps/puppetserver/code-
content-command_script.sh
}

```

auth.conf

Puppet Server's `auth.conf` file contains rules for authorizing access to Puppet Server's HTTP API endpoints. For an overview, see [Configuring Puppet Server](#) on page 72.

The rules are defined in a file named `auth.conf`, and Puppet Server applies the settings when a request's endpoint matches a rule.

Note: You can also use the [puppetlabs-puppet_authorization](#) module to manage the new `auth.conf` file's authorization rules in the new HOCON format, and the [puppetlabs-hocon](#) module to use Puppet to manage HOCON-formatted settings in general.

To configure how Puppet Server authenticates requests, use the supported HOCON `auth.conf` file and authorization methods, and see the parameters and rule definitions in the [HOCON Parameters](#) section.

You can find the Puppet Server `auth.conf` file [here](#).

HOCON example

Here is an example authorization section using the HOCON configuration format:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path: "^/my_path/([^/]+)$"
        type: regex
        method: get
      }
      allow: [ node1, node2, node3, {extensions:{ext_shortname1:
value1, ext_shortname2: value2}} ]
      sort-order: 1
      name: "user-specific my_path"
    },
    {
      match-request: {
        path: "/my_other_path"
        type: path
      }
      allow-unauthenticated: true
      sort-order: 2
      name: "my_other_path"
    },
  ]
}
```

For a more detailed example of how to use the HOCON configuration format, see [Configuring The Authorization Service](#).

For descriptions of each setting, see the following sections.

HOCON parameters

Use the following parameters when writing or migrating custom authorization rules using the new HOCON format.

version

The `version` parameter is required. In this initial release, the only supported value is 1.

allow-header-cert-info

Note: Puppet Server ignores the setting of the same name in [puppetserver.conf](#) on page 75 in favor of this setting in the `auth.conf` file.

This optional `authorization` section parameter determines whether to enable [External SSL termination](#) on page 142 on all HTTP endpoints that Puppet Server handles, including the Puppet API, the certificate authority API, and the Puppet Admin API. It also controls how Puppet Server derives the user's identity for authorization purposes. The default value is `false`.

If this setting is `true`, Puppet Server ignores any presented certificate and relies completely on header data to authorize requests.

Warning! This is very insecure; **do not enable this parameter** unless you've secured your network to prevent **any** untrusted access to Puppet Server.

You cannot rename any of the X-Client headers when this setting is enabled, and you must specify identity through the X-Client-Verify, X-Client-DN, and X-Client-Cert headers.

For more information, see [Disable HTTPS for Puppet Server](#) on page 142 in the Puppet Server documentation and [Configuring the Authorization Service](#) in the `trapperkeeper-authorization` documentation.

rules

The required `rules` array of a Puppet Server's HOCON `auth.conf` file determines how Puppet Server responds to a request. Each element is a map of settings pertaining to a rule, and when Puppet Server receives a request, it evaluates that request against each rule looking for a match.

You define each rule by adding parameters to the rule's `match-request` section. A `rules` array can contain as many rules as you need, each with a single `match-request` section.

If a request matches a rule in a `match-request` section, Puppet Server determines whether to allow or deny the request using the `rules` parameters that follow the rule's `match-request` section:

- At least one of:
 - `allow`
 - `allow-unauthenticated`
 - `deny`
- `sort-order` (required)
- `name` (required)

If no rule matches, Puppet Server denies the request by default and returns an HTTP 403/Forbidden response.

match-request

A `match-request` can take the following parameters, some of which are required:

- **path and type (required):** A `match-request` rule must have a `path` parameter, which returns a match when a request's endpoint URL starts with or contains the `path` parameter's value. The parameter can be a literal string or regular expression as defined in the required `type` parameter.

```
# Regular expression to match a path in a URL.
path: "^/puppet/v3/report/([^/]+)$"
type: regex

# Literal string to match the start of a URL's path.
path: "/puppet/v3/report/"
type: path
```

Note: While the HOCON format doesn't require you to wrap all string values with double quotation marks, some special characters commonly used in regular expressions --- such as `*` --- break HOCON parsing unless the entire value is enclosed in double quotes.

- **method:** If a rule contains the optional `method` parameter, Puppet Server applies that rule only to requests that use its value's listed HTTP methods. This parameter's valid values are `get`, `post`, `put`, `delete`, and `head`, provided either as a single value or array of values.

```
# Use GET and POST.
method: [get, post]

# Use PUT.
method: put
```

Note: While the new HOCON format does not provide a direct equivalent to the [Deprecated features](#) on page 102 `method` parameter's `search` indirector, you can create the equivalent rule by passing `GET` and `POST` to `method` and specifying endpoint paths using the `path` parameter.

- **query-params:** Use the optional query-params setting to provide the list of query parameters. Each entry is a hash of the param name followed by a list of its values.

For example, this rule would match a request URL containing the `environment=production` or `environment=test` query parameters:

```
``` hocon
query-params: {
 environment: [production, test]
}
```
```

`allow`, `allow-unauthenticated`, and `deny`

After each rule's `match-request` section, it must also have an `allow`, `allow-unauthenticated`, or `deny` parameter. (You can set both `allow` and `deny` parameters for a rule, though Puppet Server always prioritizes `deny` over `allow` when a request matches both.)

If a request matches the rule, Puppet Server checks the request's authenticated "name" (see [allow-header-cert-info](#)) against these parameters to determine what to do with the request.

- **allow-unauthenticated:** If this Boolean parameter is set to `true`, Puppet Server allows the request --- even if it can't determine an authenticated name. **This is a potentially insecure configuration** --- be careful when enabling it. A rule with this parameter set to `true` can't also contain the `allow` or `deny` parameters.
- **allow:** This parameter can take a single string value, an array of string values, a single map value with either an `extensions` or `certname` key, or an array of string and map values.

The string values can contain:

- An exact domain name, such as `www.example.com`.
- A glob of names containing a `*` in the first segment, such as `*.example.com` or simply `*`.
- A regular expression surrounded by `/` characters, such as `/example/`.
- A backreference to a regular expression's capture group in the `path` value, if the rule also contains a `type` value of `regex`. For example, if the path for the rule were `"^/example/([^\s/]+)$"`, you can make a backreference to the first capture group using a value like `$1.domain.org`.

The map values can contain:

- An `extensions` key that specifies an array of matching X.509 extensions. Puppet Server authenticates the request only if each key in the map appears in the request, and each key's value exactly matches.
- A `certname` key equivalent to a bare string.

If the request's authenticated name matches the parameter's value, Puppet Server allows it.

Note: If you are using Puppet Server with the CA disabled, you must use OID values for the extensions. Puppet Server will not be able to resolve [short names](#) in this mode.

- **deny:** This parameter can take the same types of values as the `allow` parameter, but refuses the request if the authenticated name matches --- even if the rule contains an `allow` value that also matches.

Also, in the HOCON Puppet Server authentication method, there is no directly equivalent behavior to the [Deprecated features](#) on page 102 `auth` parameter's `on` value.

`sort-order`

After each rule's `match-request` section, the required `sort-order` parameter sets the order in which Puppet Server evaluates the rule by prioritizing it on a numeric value between 1 and 399 (to be evaluated before default Puppet rules) or 601 to 998 (to be evaluated after Puppet), with lower-numbered values evaluated first. Puppet Server secondarily sorts rules lexicographically by the name string value's Unicode code points.

```
sort-order: 1
```

`name`

After each rule's `match-request` section, this required parameter's unique string value identifies the rule to Puppet Server. The name value is also written to server logs and error responses returned to unauthorized clients.

```
name: "my path"
```

Note: If multiple rules have the same name value, Puppet Server will fail to launch.

webserver.conf

The `webserver.conf` file configures the Puppet Server `webserver` service. For an overview, see [Configuring Puppet Server](#) on page 72. To configure the mount points for the Puppet administrative API web applications, see the [web-routes.conf](#) on page 82.

Examples

The `webserver.conf` file looks something like this:

```
# Configure the webserver.
webserver: {
  # Log webserver access to a specific file.
  access-log-config: /etc/puppetlabs/puppetserver/request-logging.xml
  # Require a valid certificate from the client.
  client-auth: need
  # Listen for HTTPS traffic on all available hostnames.
  ssl-host: 0.0.0.0
  # Listen for HTTPS traffic on port 8140.
  ssl-port: 8140
}
```

These are the main values for managing a Puppet Server installation. For further documentation, including a complete list of available settings and values, see [Configuring the Webserver Service](#).

By default, Puppet Server is configured to use the correct Puppet primary server and certificate authority (CA) certificates. If you're using an external CA and providing your own certificates and keys, make sure the SSL-related parameters in `webserver.conf` point to the correct file.

```
webserver: {
  ...
  ssl-cert      : /path/to/server.pem
  ssl-key       : /path/to/server.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
}
```

web-routes.conf

The `web-routes.conf` file configures the Puppet Server `web-router-service`, which sets mount points for Puppet Server's web applications. You should not modify these mount points, as Puppet 4 agents rely on Puppet Server mounting them to specific URLs.

For an overview, see [Configuring Puppet Server](#) on page 72. To configure the `webserver` service, see the [webserver.conf](#) on page 82.

Example

The `web-routes.conf` file looks like this:

```
# Configure the mount points for the web apps.
web-router-service: {
  # These two should not be modified because the Puppet 4 agent expects
  them to
  # be mounted at these specific paths.
```

```

    "puppetlabs.services.ca.certificate-authority-service/certificate-
    authority-service": "/puppet-ca"
    "puppetlabs.services.master.master-service/master-service": "/puppet"
    "puppetlabs.services.legacy-routes.legacy-routes-service/legacy-routes-
    service": ""

    # This controls the mount point for the Puppet administration API.
    "puppetlabs.services.puppet-admin.puppet-admin-service/puppet-admin-
    service": "/puppet-admin-api"
  }

```

global.conf

The `global.conf` file contains global configuration settings for Puppet Server. For an overview, see [Configuring Puppet Server](#) on page 72.

You shouldn't typically need to make changes to this file. However, you can change the `logging-config` path for the logback logging configuration file if necessary. For more information about the logback file, see <http://logback.qos.ch/manual/configuration.html>.

Example

```

global: {
  logging-config: /etc/puppetlabs/puppetserver/logback.xml
}

```

ca.conf

The `ca.conf` file configures settings for the Puppet Server Certificate Authority (CA) service. For an overview, see [Configuring Puppet Server](#) on page 72.

Signing settings

The `allow-subject-alt-names` setting in the `certificate-authority` section enables you to sign certs with subject alternative names. It is false by default for security reasons, but can be enabled if you need to sign certs with subject alternative names. `puppet cert sign` used to allow this via a flag, but `puppetserver ca sign` requires it to be configured in the config file.

The `allow-authorization-extensions` setting in the `certificate-authority` section enables you to sign certs with authorization extensions. It is false by default for security reasons, but can be enabled if you know you need to sign certs this way. `puppet cert sign` used to allow this via a flag, but `puppetserver ca sign` requires it to be configured in the config file.

Infrastructure CRL settings

Puppet Server is able to create a separate CRL file containing only revocations of Puppet infrastructure nodes. This behavior is turned off by default. To enable it, set `certificate-authority.enable-infra-crl` to `true`.

product.conf

The `product.conf` file contains settings that determine how Puppet Server interacts with Puppet, Inc., such as automatic update checking and analytics data collection.

Settings

The `product.conf` file doesn't exist in a default Puppet Server installation; to configure its settings, you must create it in Puppet Server's `conf.d` directory (located by default at `/etc/puppetlabs/puppetserver/conf.d`). This file is a [HOCON-formatted](#) configuration file with the following settings:

- Settings in the `product` section configure update checking and analytics data collection:
 - `check-for-updates`: If set to `false`, Puppet Server will not automatically check for updates, and will not send analytics data to Puppet.

If this setting is unspecified (default) or set to `true`, Puppet Server checks for updates upon start or restart, and every 24 hours thereafter, by sending the following data to Puppet:

- Product name
- Puppet Server version
- IP address
- Data collection timestamp

Puppet requests this data as one of the many ways we learn about and work with our community. The more we know about how you use Puppet, the better we can address your needs. No personally identifiable information is collected, and the data we collect is never used or shared outside of Puppet.

Example

```
# Disabling automatic update checks and corresponding analytic data
collection

product: {
  check-for-updates: false
}
```

logback.xml

Puppet Server's logging is routed through the Java Virtual Machine's [Logback library](#) and configured in an XML file typically named `logback.xml`.

Note: This document covers basic, commonly modified options for Puppet Server logs. Logback is a powerful library with many options. For detailed information on configuring Logback, see the [Logback Configuration Manual](#).

For advanced logging configuration tips specific to Puppet Server, such as configuring Logstash or outputting logs in JSON format, see [Advanced logging configuration](#) on page 85.

Puppet Server logging

By default, Puppet Server logs messages and errors to `/var/log/puppetlabs/puppetserver/puppetserver.log`. The default log level is 'INFO', and Puppet Server sends nothing to syslog. You can change Puppet Server's logging behavior by editing `/etc/puppetlabs/puppetserver/logback.xml`, and you can specify a different Logback config file in `global.conf`.

You can restart the `puppetserver` service for changes to take effect, or enable [configuration scanning](#) to allow changes to be recognized at runtime.

Puppet Server also relies on Logback to manage, rotate, and archive Server log files. Logback archives Server logs when they exceed 10MB, and when the total size of all Server logs exceeds 1GB, it automatically deletes the oldest logs.

Settings

level

To modify Puppet Server's logging level, change the `level` attribute of the `root` element. By default, the logging level is set to `info`:

```
<root level="info">
```

Supported logging levels, in order from most to least information logged, are `trace`, `debug`, `info`, `warn`, and `error`. For instance, to enable debug logging for Puppet Server, change `info` to `debug`:

```
<root level="debug">
```

Puppet Server profiling data is included at the debug logging level.

You can also change the logging level for JRuby logging from its defaults of `error` and `info` by setting the `level` attribute of the `jruby` element. For example, to enable debug logging for JRuby, set the attribute to `debug`:

```
<jruby level="debug">
```

Logging location

You can change the file to which Puppet Server writes its logs in the appender section named `F1`. By default, the location is set to `/var/log/puppetlabs/puppetserver/puppetserver.log`:

```
...
  <appender name="F1" class="ch.qos.logback.core.FileAppender">
    <file>/var/log/puppetlabs/puppetserver/puppetserver.log</file>
  ...
```

To change this to `/var/log/puppetserver.log`, modify the contents of the `file` element:

```
<file>/var/log/puppetserver.log</file>
```

The user account that owns the Puppet Server process must have write permissions to the destination path.

scan and scanPeriod

Logback supports noticing and reloading configuration changes without requiring a restart, a feature Logback calls **scanning**. To enable this, set the `scan` and `scanPeriod` attributes in the `<configuration>` element of `logback.xml`:

```
<configuration scan="true" scanPeriod="60 seconds">
```

Due to a [bug in Logback](#), the `scanPeriod` must be set to a value; setting only `scan="true"` will not enable configuration scanning. Scanning is enabled by default in the `logback.xml` configuration packaged with Puppet Server.

Note: The HTTP request log does not currently support the scan feature. Adding the `scan` or `scanPeriod` settings to `request-logging.xml` will have no effect.

HTTP request logging

Puppet Server logs HTTP traffic separately, and this logging is configured in a different Logback configuration file located at `/etc/puppetlabs/puppetserver/request-logging.xml`. To specify a different Logback configuration file, change the `access-log-config` setting in Puppet Server's [webservice.conf](#) on page 82 file.

The HTTP request log uses the same Logback configuration format and settings as the Puppet Server log. It also lets you configure what it logs using patterns, which follow Logback's [PatternLayout](#) format.

Advanced logging configuration

Puppet Server uses the [Logback](#) library to handle all of its logging. Logback configuration settings are stored in the [logback.xml](#) on page 84 file, which is located at `/etc/puppetlabs/puppetserver/logback.xml` by default.

You can configure Logback to log messages in JSON format, which makes it easy to send them to other logging backends, such as Logstash.

Configuring Puppet Server for use with Logstash

There are a few steps necessary to setup your Puppet Server logging for use with Logstash. The first step is to modify your logging configuration so that Puppet Server is logging in a JSON format. After that, you'll configure an external tool to monitor these JSON files and send the data to Logstash (or another remote logging system).

Configuring Puppet Server to log to JSON

Before you configure Puppet Server to log to JSON, consider the following:

- Do you want to configure Puppet Server to *only* log to JSON, instead of the default plain-text logging? Or do you want to have JSON logging *in addition to* the default plain-text logging?
- Do you want to set up JSON logging *only* for the main Puppet Server logs (`puppetserver.log`), or *also* for the HTTP access logs (`puppetserver-access.log`)?
- What kind of log rotation strategy do you want to use for the new JSON log files?

The following examples show how to configure Logback for:

- logging to both JSON and plain-text
- JSON logging both the main logs and the HTTP access logs
- log rotation on the JSON log files

Adjust the example configuration settings to suit your needs.

Note: Puppet Server also relies on Logback to manage, rotate, and archive Server log files. Logback archives Server logs when they exceed 200MB, and when the total size of all Server logs exceeds 1GB, it automatically deletes the oldest logs.

Adding a JSON version of the main Puppet Server logs

Logback writes logs using components called [appenders](#). The example code below uses `RollingFileAppender` to rotate the log files and avoid consuming all of your storage.

1. To configure Puppet Server to log its main logs to a second log file in JSON format, add an appender section like the following example to your `logback.xml` file, at the same level in the XML as existing appenders. The order of the appenders does not matter.

```
<appender name="JSON"
  class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>/var/log/puppetlabs/puppetserver/puppetserver.log.json</file>

  <rollingPolicy
    class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
    <fileNamePattern>/var/log/puppetlabs/puppetserver/
puppetserver.log.json.%d{yyyy-MM-dd}</fileNamePattern>
    <maxHistory>5</maxHistory>
    </rollingPolicy>

    <encoder class="net.logstash.logback.encoder.LogstashEncoder"/>
  </appender>
```

2. Activate the appended by adding an `appender-ref` entry to the `<root>` section of `logback.xml`:

```
<root level="info">
  <appender-ref ref="FILE"/>
  <appender-ref ref="JSON"/>
</root>
```

3. If you decide you want to log *only* the JSON format, comment out the other `appender-ref` entries.

`LogstashEncoder` has many configuration options, including the ability to modify the list of fields that you want to include, or give them different field names. For more information, see the [Logstash Logback Encoder Docs](#).

Adding a JSON version of the Puppet Server HTTP Access logs

To add JSON logging for HTTP requests:

1. Add the following Logback appender section to the `request-logging.xml` file:

```
{% raw %}
<appender name="JSON"
  class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>/var/log/puppetlabs/puppetserver/puppetserver-access.log.json</file>

  <rollingPolicy
    class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
    <fileNamePattern>/var/log/puppetlabs/puppetserver/puppetserver-access.log.json.%d{yyyy-MM-dd}</fileNamePattern>
    <maxHistory>30</maxHistory>
  </rollingPolicy>

  <encoder
    class="net.logstash.logback.encoder.AccessEventCompositeJsonEncoder">
    <providers>
      <version/>
      <pattern>
        <pattern>
          {
            "@timestamp":"%date{yyyy-MM-dd'T'HH:mm:ss.SSSXXX}" ,
            "clientip":"%remoteIP" ,
            "auth":"%user" ,
            "verb":"%requestMethod" ,
            "requestprotocol":"%protocol" ,
            "rawrequest":"%requestURL" ,
            "response":"#asLong{%statusCode}" ,
            "bytes":"#asLong{%bytesSent}" ,
            "total_service_time":"#asLong{%elapsedTime}" ,
            "request":"http://%header{Host}%requestURI" ,
            "referrer":"%header{Referer}" ,
            "agent":"%header{User-agent}" ,

            "request.host":"%header{Host}" ,
            "request.accept":"%header{Accept}" ,
            "request.accept-encoding":"%header{Accept-
Encoding}" ,

            "request.connection":"%header{Connection}" ,

            "puppet.client-verify":"%header{X-Client-Verify}" ,
            "puppet.client-dn":"%header{X-Client-DN}" ,
            "puppet.client-cert":"%header{X-Client-Cert}" ,

            "response.content-type":"%responseHeader{Content-
Type}" ,

            "response.content-length":"%responseHeader{Content-
Length}" ,

            "response.server":"%responseHeader{Server}" ,
            "response.connection":"%responseHeader{Connection}"
          }
        </pattern>
      </pattern>
    </providers>
  </encoder>
</appender>
{% endraw %}
```

2. Add a corresponding `appender-ref` in the configuration section:

```
<appender-ref ref="JSON"/>
```

For more information about options available for the `pattern` section, see the [Logback Logstash Encoder Docs](#).

Sending the JSON data to Logstash

After configuring Puppet Server to log messages in JSON format, you must also configure it to send the logs to Logstash (or another external logging system). There are several different ways to approach this:

- Configure Logback to send the data to Logstash directly, from within Puppet Server. See the Logstash-Logback encoder docs on how to send the logs by [TCP](#) or [UDP](#). Note that TCP comes with the risk of bottlenecking Puppet Server if your Logstash system is busy, and UDP might silently drop log messages.
- [Filebeat](#) is a tool from Elastic for shipping log data to Logstash.
- [Logstash Forwarder](#) is an earlier tool from Elastic with similar capabilities.

Adding file server mount points

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

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

To set up a mount point:

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

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

Mount points in the Puppet URI

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

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

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

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

- The special `locales` mount point serves files from the `locales` directory of every module to support automatic downloading of module translations to agents. It also behaves like the `plugins` mount point, and also has a different source directory.

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

Creating a new mount point in `fileserver.conf`

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

```
[<NAME OF MOUNT POINT>]
  path <PATH TO DIRECTORY>
```

In the following example, a file at `/etc/puppetlabs/puppet/installer_files/oracle.pkg` would be available in manifests as `puppet:///installer_files/oracle.pkg`:

```
[installer_files]
  path /etc/puppetlabs/puppet/installer_files
```

Make sure that the `puppet` user has the right permissions to access that directory and its contents.



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

Controlling access to a custom mount point in `auth.conf`

By default, any node with a valid certificate can access the files in your new mount point. If a node can fetch a catalog, it can fetch files. If the node can't fetch a catalog, it can't fetch files. This is the same behavior as the special `modules` and `plugins` mount points. If necessary, you can restrict access to a custom mount point in `auth.conf`.

To add a new auth rule to Puppet Server's HOCON-format `auth.conf` file, located at `/etc/puppetlabs/puppetserver/conf.d/auth.conf`, you must meet the following requirements:

- It must match requests to all four of these prefixes:
 - `/puppet/v3/file_metadata/<MOUNT POINT>`
 - `/puppet/v3/file_metadatas/<MOUNT POINT>`
 - `/puppet/v3/file_content/<MOUNT POINT>`
 - `/puppet/v3/file_contents/<MOUNT POINT>`
- Its `sort-order` must be lower than 500, so that it overrides the default rule for the file server.

For example:

```
{
  # Allow limited access to files in /etc/puppetlabs/puppet/
  installer_files:
    match-request: {
      path: "^/puppet/v3/file_(content|metadata)s?/installer_files"
      type: regex
    }
    allow: "*.dev.example.com"
    sort-order: 400
    name: "dev.example.com large installer files"
},
```

Related topics: [Module fundamentals](#), [fileserver.conf: Custom fileserver mount points](#), [Puppet Server configuration files: puppetserver.conf](#), [Puppet Server configuration files: auth.conf](#).

Checking the values of settings

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

General usage

The `puppet config print` command loads and evaluates settings, and can imitate any of Puppet's other commands and services when doing so. The `--section` and `--environment` options let you control how settings are loaded; for details, see the sections below on imitating different services.

Note: To ensure that you're seeing the values Puppet use when running as a service, be sure to use `sudo` or run the command as root or Administrator. If you run `puppet config print` as some other user, Puppet might not use the [system config file](#).

To see the value of one setting:

```
sudo puppet config print <SETTING NAME> [--section <CONFIG SECTION>] [--environment <ENVIRONMENT>]
```

This displays just the value of `<SETTING NAME>`.

To see the value of multiple settings:

```
sudo puppet config print <SETTING 1> <SETTING 2> [...] [--section <CONFIG SECTION>] [--environment <ENVIRONMENT>]
```

This displays `name = value` pairs for all requested settings.

To see the value of all settings:

```
sudo puppet config print [--section <CONFIG SECTION>] [--environment <ENVIRONMENT>]
```

This displays `name = value` pairs for all settings.

Config sections

The `--section` option specifies which section of `puppet.conf` to use when finding settings. It is optional, and defaults to `main`. Valid sections are:

- `main` (default) — used by all commands and services
- `server` — used by the primary Puppet server service and the `puppetserver ca` command
- `agent` — used by the Puppet agent service
- `user` — used by the Puppet apply command and most other commands

As usual, the other sections override the `main` section if they contain a setting; if they don't, the value from `main` is used, or a default value if the setting isn't present there.

Environments

The `--environment` option specifies which [environment](#) to use when finding settings. It is optional and defaults to the value of the `environment` setting in the `user` section (usually `production`, because it's rare to specify an environment in `user`).

You can only specify environments that exist.

This option is primarily useful when looking up settings used by the primary server service, because it's rare to use environment config sections for Puppet apply and Puppet agent.

Imitating Puppet server and `puppetserver ca`

To see the settings the Puppet server service and the `puppetserver ca` command would use:

- Specify `--section server`.
- Use the `--environment` option to specify the environment you want settings for, or let it default to `production`.
- Remember to use `sudo`.
- If your primary Puppet server is managed as a Rack application (for example, with Passenger), check the `config.ru` file to make sure it's using the `confdir` and `vardir` that you expect. If it's using non-standard ones, you need to specify them on the command line with the `--confdir` and `--vardir` options; otherwise you might not see the correct values for settings.

To see the effective `modulepath` used in the `dev` environment:

```
sudo puppet config print modulepath --section server --environment dev
```

This returns something like:

```
/etc/puppetlabs/code/environments/dev/modules:/etc/puppetlabs/code/modules:/opt/puppetlabs/puppet/modules
```

To see whether PuppetDB is configured for exported resources:

```
sudo puppet config print storeconfigs storeconfigs_backend --section server
```

This returns something like:

```
storeconfigs = true
storeconfigs_backend = puppetdb
```

Imitating Puppet agent

To see the settings the Puppet agent service would use:

- Specify `--section agent`.
- Remember to use `sudo`.
- If you are seeing something unexpected, check your Puppet agent init script or cron job to make sure it is using the standard `confdir` and `vardir`, is running as root, and isn't overriding other settings with command line options. If it's doing anything unusual, you might have to set more options for the `config print` command.

To see whether the agent is configured to use manifest ordering when applying the catalog:

```
sudo puppet config print ordering --section agent
```

This returns something like:

```
manifest
```

Imitating `puppet apply`

To see the settings the Puppet apply command would use:

- Specify `--section user`.
- Remember to use `sudo`.
- If you are seeing something unexpected, check the cron job or script that is responsible for configuring the machine with Puppet apply. Make sure it is using the standard `confdir` and `vardir`, is running as root, and isn't overriding other settings with command line options. If it's doing anything unusual, you might have to set more options for the `config print` command.

To see whether Puppet apply is configured to use reports:

```
sudo puppet config print report reports --section user
```

This returns something like:

```
report = true
reports = store,http
```

Editing settings on the command line

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

Use `puppet config set` for:

- Fast one-off config changes,
- Scriptable config changes in provisioning tools,

If you find yourself changing many settings, edit the `puppet.conf` file instead, or manage it with a template.

Usage

To assign a new value to a setting, run:

```
sudo puppet config set <SETTING NAME> <VALUE> --section <CONFIG SECTION>
```

This declaratively sets the value of `<SETTING NAME>` to `<VALUE>` in the specified config section, regardless of whether the setting already had a value.

Config sections

The `--section` option specifies which section of `puppet.conf` to modify. It is optional, and defaults to `main`. Valid sections are:

- `main` (default) — used by all commands and services
- `server` — used by the primary Puppet server service and the `puppetserver ca` command
- `agent` — used by the Puppet agent service
- `user` — used by the `puppet apply` command and most other commands

When modifying the [system config file](#), use `sudo` or run the command as `root` or Administrator.

Example

Consider the following `puppet.conf` file:

```
[main]
certname = agent01.example.com
server = server.example.com
vardir = /var/opt/lib/pe-puppet

[agent]
report = true
graph = true
pluginsync = true

[server]
dns_alt_names = server,server.example.com,puppet,puppet.example.com
```

If you run the following commands:

```
sudo puppet config set reports puppetdb --section server
sudo puppet config set ordering manifest
```

The `puppet.conf` file now looks like this:

```
[main]
certname = agent01.example.com
server = server.example.com
vardir = /var/opt/lib/pe-puppet
ordering = manifest

[agent]
report = true
graph = true
pluginsync = true

[server]
dns_alt_names = server,server.example.com,server,server.example.com
reports = puppetdb
```

Differing behavior in puppet.conf

Puppet Server honors almost all settings in `puppet.conf` and should pick them up automatically. For more complete information on `puppet.conf` settings, see our [Configuration Reference](#) page.

Settings that differ

`autoflush`

Puppet Server does not use this setting. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

`bindaddress`

Puppet Server does not use this setting. To set the address on which the primary server listens, use either `host` (unencrypted) or `ssl-host` (SSL encrypted) in the [webserver.conf](#) file.

`ca`

Puppet Server does not use this setting. Instead, Puppet Server acts as a certificate authority based on the certificate authority service configuration in the `ca.cfg` file. See [Service Bootstrapping](#) on page 74 for more details.

`ca_ttl`

Puppet Server enforces a max ttl of 50 standard years (up to 1576800000 seconds).

`cacert`

If you enable Puppet Server's certificate authority service, it uses the `cacert` setting in `puppet.conf` to determine the location of the CA certificate for such tasks as generating the CA certificate or using the CA to sign client certificates. This is true regardless of the configuration of the `ssl-` settings in [webserver.conf](#).

`cacrl`

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, or `ssl-crl-path` in [webserver.conf](#), Puppet Server uses the file at `ssl-crl-path` as the CRL for authenticating clients via SSL. If at least one of the `ssl-` settings in `webserver.conf` is set but `ssl-crl-path` is not set, Puppet Server will *not* use a CRL to validate clients via SSL.

If none of the `ssl-` settings in `webserver.conf` are set, Puppet Server uses the CRL file defined for the `hostcrl` setting---and not the file defined for the `cacrl` setting---in `puppet.conf`. At start time, Puppet Server copies the file for the `cacrl` setting, if one exists, over to the location in the `hostcrl` setting.

Any CRL file updates from the Puppet Server certificate authority---such as revocations performed via the `certificate_status` HTTP endpoint---use the `cacrl` setting in `puppet.conf` to determine the location of the CRL. This is true regardless of the `ssl-` settings in `webserver.conf`.

`capass`

Puppet Server does not use this setting. Puppet Server's certificate authority does not create a `capass` password file when the CA certificate and key are generated.

`caprivatedir`

Puppet Server does not use this setting. Puppet Server's certificate authority does not create this directory.

`daemonize`

Puppet Server does not use this setting.

`hostcert`

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, or `ssl-crl-path` in [webserver.conf](#), Puppet Server presents the file at `ssl-cert` to clients as the server certificate via SSL.

If at least one of the `ssl-` settings in `webserver.conf` is set but `ssl-cert` is not set, Puppet Server gives an error and shuts down at startup. If none of the `ssl-` settings in `webserver.conf` are set, Puppet Server uses the file for the `hostcert` setting in `puppet.conf` as the server certificate during SSL negotiation.

Regardless of the configuration of the `ssl-` "webserver.conf" settings, Puppet Server's certificate authority service, if enabled, uses the `hostcert` "puppet.conf" setting, and not the `ssl-cert` setting, to determine the location of the server host certificate to generate.

`hostcrl`

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, or `ssl-crl-path` in [webserver.conf](#), Puppet Server uses the file at `ssl-crl-path` as the CRL for authenticating clients via SSL. If at least one of the `ssl-` settings in `webserver.conf` is set but `ssl-crl-path` is not set, Puppet Server will *not* use a CRL to validate clients via SSL.

If none of the `ssl-` settings in `webserver.conf` are set, Puppet Server uses the CRL file defined for the `hostcrl` setting---and not the file defined for the `cacrl` setting---in `puppet.conf`. At start time, Puppet Server copies the file for the `cacrl` setting, if one exists, over to the location in the `hostcrl` setting.

Any CRL file updates from the Puppet Server certificate authority---such as revocations performed via the `certificate_status` HTTP endpoint---use the `cacrl` setting in `puppet.conf` to determine the location of the CRL. This is true regardless of the `ssl-` settings in `webserver.conf`.

`hostprivkey`

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, or `ssl-crl-path` in [webserver.conf](#), Puppet Server uses the file at `ssl-key` as the server private key during SSL transactions.

If at least one of the `ssl-` settings in `webserver.conf` is set but `ssl-key` is not, Puppet Server gives an error and shuts down at startup. If none of the `ssl-` settings in `webserver.conf` are set, Puppet Server uses the file for the `hostprivkey` setting in `puppet.conf` as the server private key during SSL negotiation.

If you enable the Puppet Server certificate authority service, Puppet Server uses the `hostprivkey` setting in `puppet.conf` to determine the location of the server host private key to generate. This is true regardless of the configuration of the `ssl-` settings in `webserver.conf`.

`http_debug`

Puppet Server does not use this setting. Debugging for HTTP client code in Puppet Server is controlled through Puppet Server's common logging mechanism. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

`keylength`

Puppet Server does not currently use this setting. Puppet Server's certificate authority generates 4096-bit keys in conjunction with any SSL certificates that it generates.

localcacert

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, and/or `ssl-crl-path` in `webserver.conf`, Puppet Server uses the file at `ssl-ca-cert` as the CA cert store for authenticating clients via SSL.

If at least one of the `ssl-` settings in `webserver.conf` is set but `ssl-ca-cert` is not set, Puppet Server gives an error and shuts down at startup. If none of the `ssl-` settings in `webserver.conf` is set, Puppet Server uses the CA file defined for the `localcacert` setting in `puppet.conf` for SSL authentication.

logdir

Puppet Server does not use this setting. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

masterhttplog

Puppet Server does not use this setting. You can configure a web server access log via the `access-log-config` setting in the `webserver.conf` file.

masterlog

Puppet Server does not use this setting. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

masterport

Puppet Server does not use this setting. To set the port on which the primary server listens, set the `port` (unencrypted) or `ssl-port` (SSL encrypted) setting in the `webserver.conf` file.

puppetdlog

Puppet Server does not use this setting. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

rails_loglevel

Puppet Server does not use this setting.

railslog

Puppet Server does not use this setting.

ssl_client_header

Puppet Server honors this setting only if the `allow-header-cert-info` setting in the `server.conf` file is set to 'true'. For more information on this setting, see the documentation on [External SSL termination](#) on page 142.

ssl_client_verify_header

Puppet Server honors this setting only if the `allow-header-cert-info` setting in the `server.conf` file is set to true. For more information on this setting, see the documentation on [External SSL termination](#) on page 142.

ssl_server_ca_auth

Puppet Server does not use this setting. It only considers the `ssl-ca-cert` setting from the `webserver.conf` file and the `cacert` setting from the `puppet.conf` file. See [cacert](#) for more information.

syslogfacility

Puppet Server does not use this setting.

user

Puppet Server does not use this setting.

HttpPool-Related Server Settings

configtimeout

Puppet Server does not currently consider this setting when making HTTP requests. This pertains, for example, to any requests that the primary server would make to the `reporturl` for the `http` report processor.

`http_proxy_host`

Puppet Server does not currently consider this setting when making HTTP requests. This pertains, for example, to any requests that the primary server would make to the `reporturl` for the `http` report processor.

`http_proxy_port`

Puppet Server does not currently consider this setting when making HTTP requests. This pertains, for example, to any requests that the primary server would make to the `reporturl` for the `http` report processor.

Overriding Puppet settings in Puppet Server

Currently, the `jruby-puppet` section of your `puppetserver.conf` file contains five settings (`master-conf-dir`, `master-code-dir`, `master-var-dir`, `master-run-dir`, and `master-log-dir`) that allow you to override settings set in your `puppet.conf` file. On installation, these five settings will be set to the proper default values.

While you are free to change these settings at will, please note that any changes made to the `master-conf-dir` and `master-code-dir` settings absolutely **MUST** be made to the corresponding Puppet settings (`confdir` and `codedir`) as well to ensure that Puppet Server and the Puppet CLI tools (such as `puppetserver ca` and `puppet` module) use the same directories. The `master-conf-dir` and `master-code-dir` settings apply to Puppet Server only, and will be ignored by the ruby code that runs when the Puppet CLI tools are run.

For example, say you have the `codedir` setting left unset in your `puppet.conf` file, and you change the `master-code-dir` setting to `/etc/my-puppet-code-dir`. In this case, Puppet Server will read code from `/etc/my-puppet-code-dir`, but the `puppet` module tool will think that your code is stored in `/etc/puppetlabs/code`.

While it is not as critical to keep `master-var-dir`, `master-run-dir`, and `master-log-dir` in sync with the `vardir`, `rundir`, and `logdir` Puppet settings, please note that this applies to these settings as well.

Also, please note that these configuration differences also apply to the interpolation of the `confdir`, `codedir`, `vardir`, `rundir`, and `logdir` settings in your `puppet.conf` file. So, take the above example, wherein you set `master-code-dir` to `/etc/my-puppet-code-dir`. Because the `basemodulepath` setting is by default `$codedir/modules:/opt/puppetlabs/puppet/modules`, then Puppet Server would use `/etc/my-puppet-code-dir/modules:/opt/puppetlabs/puppet/modules` for the value of the `basemodulepath` setting, whereas the `puppet` module tool would use `/etc/puppetlabs/code/modules:/opt/puppetlabs/puppet/modules` for the value of the `basemodulepath` setting.

Components and functionality

Puppet is made up of several packages. Together these are called the Puppet platform, which is what you use to manage, store and run your Puppet code. These packages include `puppetserver`, `puppetdb`, and `puppet-agent` — which includes *Facter* and *Hiera*.

- [Puppet Server](#) on page 97
- [PuppetDB](#) on page 223

All of the data generated by Puppet (for example facts, catalogs, reports) is stored in PuppetDB.

- [Facter](#) on page 223

Facter is Puppet's cross-platform system profiling library. It discovers and reports per-node facts, which are available in your Puppet manifests as variables.

- [Hiera](#) on page 241

Hiera is a built-in key-value configuration data lookup system, used for separating data from Puppet code.

- [Environments](#) on page 289

Environments are isolated groups of agent nodes.

- [Important directories and files](#) on page 296

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.

- [Puppet services and tools](#) on page 305

Puppet provides a number of core services and administrative tools to manage systems with or without a primary Puppet server, and to compile configurations for Puppet agents.

- [Classifying nodes](#) on page 329

You can classify nodes using an external node classifier (ENC), which is a script or application that tells Puppet which classes a node must have. It can replace or work in concert with the node definitions in the main site manifest (`site.pp`).

- [Puppet reports](#) on page 332

Puppet creates a report about its actions and your infrastructure each time it applies a catalog during a Puppet run. You can create and use report processors to generate insightful information or alerts from those reports.

- [Built-in report processors](#)

- [Puppet's internals](#) on page 341

Learn the details of Puppet's internals, including how primary servers and agents communicate via host-verified HTTPS, and about the process of catalog compilation.

Puppet Server

- [About Puppet Server](#) on page 97
- [Puppet Server release notes](#) on page 34

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

- [Deprecated features](#) on page 102
- [Compatibility with Puppet agent](#) on page 107
- [Installing Puppet Server](#) on page 107
- [Differing behavior in puppet.conf](#) on page 93
- [Puppet Server HTTP API](#) on page 172
- [Certificate Clean](#) on page 192
- [Certificate authority and SSL](#) on page 203

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.

About Puppet Server

Puppet is configured in an agent-server architecture, in which a primary server node manages the configuration information for a fleet of agent nodes. Puppet Server acts as the primary server node.

Puppet Server is a Ruby and Clojure application that runs on the Java Virtual Machine (JVM). Puppet Server runs Ruby code for compiling Puppet catalogs and for serving files in several JRuby interpreters. It also provides a certificate authority through Clojure.

This page describes the general requirements and the run environment for Puppet Server.

Supported Platforms

Puppet provides Puppet Server packages for the following platforms:

- Red Hat Enterprise Linux
- Centos
- Debian
- Ubuntu

If we do not provide a package for your system, you can run Puppet Server from source on any x86_64 Linux server with JDK 1.8 or 11. See [Running from source](#) on page 166 for more details.

Tip: Other POSIX compatible servers or versions of Java are unsupported by Puppet, but they may work. Join our [community Slack](#) for help with non-supported operating systems, architectures, or JRE versions.

Puppet Server releases

Puppet Server and Puppet share the same major release (Puppet Server 6.x and Puppet 6.x). However, they are versioned separately and might have different minor or patch versions (Puppet Server 6.5 versus Puppet 6.8). For a list of the maintained versions of Puppet, Puppet Server, and Puppet DB, see [Puppet releases and lifecycles](#).

Controlling the Service

The Puppet Server service name is `puppetserver`. To start and stop the service, use commands such as `service puppetserver restart`, `service puppetserver status` for your OS.

Puppet Server's Run Environment

Puppet Server consists of several related services. These services share state and route requests among themselves. The services run inside a single JVM process, using the Trapperkeeper service framework.

Embedded Web Server

Puppet Server uses a Jetty-based web server embedded in the service's JVM process. No additional or unique actions are required to configure and enable the web server.

You can modify the web server's settings in [webservice.conf](#) on page 82. You might need to edit this file if you use an external CA or run Puppet on a non-standard port.

Puppet API Service

Puppet Server provides APIs that are used by the Puppet agent to manage the configuration of your nodes. See [Puppet V3 HTTP API](#) for more information on the basic APIs.

Certificate Authority Service

Puppet Server includes a certificate authority (CA) service that:

- Accepts certificate signing requests (CSRs) from nodes.
- Serves certificates and a certificate revocation list (CRL) to nodes.
- Optionally accepts commands to sign or revoke certificates.

See [CA V1 HTTP API](#) for more information on these APIs.

Signing and revoking certificates over the network is disabled by default. You can use the `auth.conf` file to allow specific certificate owners the ability to issue commands.

The CA service stores credentials using `.pem` files. You can use the `puppetserver ca` command to interact with these credentials, including listing, signing, and revoking certificates.

Admin API Service

Puppet Server includes an administrative API for triggering maintenance tasks. The most common task refreshes Puppet's environment cache, which causes all of your Puppet code to reload without the requirement to restart the service. Consequently, you can deploy new code to long-timeout environments without executing a full restart of the service.

- For API docs, see:
 - [Environment cache](#) on page 202
 - [JRuby pool](#) on page 202
- For details about environment caching, see [About environments](#).

JRuby Interpreters

Most of Puppet Server's work is done by Ruby code running in JRuby. JRuby is an implementation of the Ruby interpreter that runs on the JVM.

You cannot use the system gem command to install Ruby Gems for the Puppet primary server. Instead, Puppet Server includes a separate `puppetserver gem` command for installing any libraries your Puppet extensions might require. See [Using Ruby gems](#) on page 138 for details.

If you want to test or debug code to be used by the Puppet Server, you can use the `puppetserver ruby` and `puppetserver irb` commands to execute Ruby code in a JRuby environment.

To handle parallel requests from agent nodes, Puppet Server maintains separate JRuby interpreters. These JRuby interpreters individually run Puppet's application code, and distribute agent requests among them.

You can configure the JRuby interpreters in the `jruby-puppet` section of [puppetserver.conf](#) on page 75.

Tuning Guide

You can maximize Puppet Server's performance by tuning your JRuby configuration. To learn more, see the Puppet Server [Tuning guide](#) on page 154.

User

Puppet Server needs to run as the user:

- `pe-puppet` if you are running Puppet Enterprise.
- `puppet` if you are running open source Puppet.

The user is specified in:

- `/etc/sysconfig/pe-puppetserver` for Puppet Enterprise.
- `/etc/sysconfig/puppetserver` for open source Puppet.

Note: Puppet Server ignores the `user` and `group` settings from `puppet.conf`.

All of the Puppet Server's files and directories must be readable and writable by this user.

Ports

By default, Puppet's HTTPS traffic uses port 8140. The OS and firewall must allow Puppet Server's JVM process to accept incoming connections on port 8140.

You can change the port in `webserver.conf` if necessary. See the [webserver.conf](#) on page 82 page for details.

Logging

All of Puppet Server's logging is routed through the JVM [Logback](#) library. By default, it logs to `/var/log/puppetlabs/puppetserver/puppetserver.log`. The default log level is 'INFO'. By default, Puppet Server sends nothing to syslog.

All log messages follow the same path, including HTTP traffic, catalog compilation, certificate processing, and all other parts of Puppet Server's work.

Puppet Server also relies on Logback to manage, rotate, and archive Server log files. Logback archives Server logs when they exceed 200MB. Also, when the total size of all Server logs exceeds 1GB, Logback automatically deletes the oldest logs.

Logback is heavily configurable. If you need something more specialized than a unified log file, it may be possible to obtain. See [Logging](#) on page 74 for more details.

Finally, any errors that happen before logging is set up or cause the logging system to die, displays in `journalctl`.

SSL Termination

By default, Puppet Server handles SSL termination automatically.

For network configurations that require external SSL termination (e.g. with a hardware load balancer), additional configuration is required. See the [External SSL termination](#) on page 142 page for details. In summary, you must:

- Configure Puppet Server to use HTTP instead of HTTPS.
- Configure Puppet Server to accept SSL information via insecure HTTP headers.
- Secure your network so that Puppet Server **cannot** be directly reached by **any** untrusted clients.
- Configure your SSL terminating proxy to set the following HTTP headers:
 - X-Client-Verify (mandatory).
 - X-Client-DN (mandatory for client-verified requests).
 - X-Client-Cert (optional; required for [trusted facts](#)).

Configuring Puppet Server

Puppet Server uses a combination of Puppet's configuration files along with its own configuration files, which are located in the `conf.d` directory. Refer to the [Config directory](#) for a list of Puppet's configuration files.

Puppet Server's `conf.d` directory contains:

- `global.conf`: Global configuration settings for Puppet Server.
- `webserver.conf` and `web-routes.conf`: Web server configuration settings.
- `puppetserver.conf`: Settings for Puppet Server itself, including the JRuby interpreter and the administrative API.
- `auth.conf`: Authentication rules for Puppet Server endpoints.
- `ca.conf`: Settings for the Certificate Authority service.

For detailed information about Puppet Server settings and the `conf.d` directory, refer to the [Configuring Puppet Server](#) on page 72 page.

Puppet Server uses Puppet's config files, including most of the settings in `puppet.conf`. However, Puppet Server treats some `puppet.conf` settings differently. You must be aware of these differences. You can view a complete list of these differences on the [Differing behavior in puppet.conf](#) on page 93 page.

Puppet Server release notes

Puppet Server 7.1.0

Released 16 March 2021.

Enhancements

- Puppet Server now adds an extension for subject-alternative-name (SAN) when it signs incoming certificate signing requests (CSR). The SAN extension contains the common name (CN) as a dns-name on the certificate. If the CSR comes with its own SAN extension, Puppet Server signs it and ensures the SAN extension includes the CSR's CN. [SERVER-2338](#)

Bug fixes

- The Jetty webserver now uses the local copy of the CRL from Puppet's SSL directory instead of the CA's copy. This fix makes it easier to set up compilers, which always have a disabled CA service and no CRL at the CA path. [SERVER-2558](#)

Deprecations

- The `master-conf-dir`, `master-code-dir`, `master-var-dir`, `master-log-dir`, and `master-run-dir` configuration settings have been deprecated in favor of `server-conf-dir`, `server-code-dir`, `server-var-dir`, `server-log-dir`, and `server-run-dir` respectively. The configuration files — which use the new settings — are shipped with the 7.1.0 `puppetserver` package. Note that the old settings are still honored for backwards compatibility, but we recommend you upgrade to the new settings. [SERVER-2867](#)

Puppet Server 7.0.3

Released 9 February 2021.

This release updates dependencies to include security fixes.

Puppet Server 7.0.2

Released 20 January 2021

Bug fix

- The warning issued when the CA dir is inside the SSL dir now only prints server logs at startup and when using the `puppetserver ca` CLI, instead of any time a Puppet command is used. ([SERVER-2934](#))

Puppet Server 7.0.1

Released 15 December 2020.

Enhancements

- The JRuby version has been bumped from 9.2.13.0 to 9.2.14.0. ([SERVER-2925](#))

Bug fixes

- The CA command line tool now correctly honors the `server` sections in the `puppet.conf`.
- When creating the symlink between the new and legacy cads the symlink will now be properly owned by the `puppet` user. ([SERVER-2917](#))

Puppet Server 7.0.0

Released 17 November 2020.

Puppet Server 7.0 is a major release. It breaks compatibility with agents prior to 4.0 and the legacy Puppet `auth.conf`, moves the default location for the `cadir`, and changes defaults for fact caching and cipher suites. See below for more details. Caution is advised when upgrading.

New features

- The default value for the `cadir` setting is now located at `/etc/puppetlabs/puppetserver/ca`. Previously, the default location was inside Puppet's own `ssldir` at `/etc/puppetlabs/puppet/ssl/ca`. This change makes it safer to delete Puppet's `ssldir` without accidentally deleting your CA certificates.
- The `puppetserver` CA CLI now provides a `migrate` command to move the CA directory from the Puppet `confdir` to the `puppetserver confdir`. It leaves behind a symlink on the old CA location, pointing to the new location at `/etc/puppetlabs/puppetserver/ca`. The symlink provides backwards compatibility for tools still expecting the `cadir` to exist in the old location. In a future release, the `cadir` setting will be removed entirely. ([SERVER-2896](#))
- The default value for the `facts cache` is now JSON instead of YAML. You can re-enable the old YAML terminus in `routes.yaml`. ([PUP-10656](#))
- Support for legacy Puppet `auth.conf` has been removed and the `jrubby-puppet.use-legacy-auth-conf` setting no longer works. Use Puppet Server's `auth.conf` file instead. ([SERVER-2778](#))
- Puppet Server no longer services requests for legacy (3.x) Puppet endpoints. Puppet Agents before 4.0 are no longer be able to check in. ([SERVER-2791](#))
- This release removes default support for many cipher suites when contacting Puppet Server. The new default supported cipher suites are: `TLS_DHE_RSA_WITH_AES_128_GCM_SHA256`, `TLS_DHE_RSA_WITH_AES_256_GCM_SHA384`, `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`, `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`, `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`, and `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`. This change aligns open source Puppet with Puppet Enterprise. Note that this change may break on old platforms. To re-enable older cipher suites you may edit the `webserver.conf`. Valid cipher suite names are listed in the [JDK Documentation](#). ([SERVER-2913](#))
- Puppet Server now provides an HTTP client whose API conforms to the HTTP client provided by Puppet. This new client is stored in the Puppet runtime as `Puppet.runtime[:http]`. ([SERVER-2780](#))

Deprecated features

The following features and configuration settings are deprecated and will be removed in a future major release of Puppet Server.

certificate-status settings

Now

If the `certificate-authority.certificate-status.authorization-required` setting is `false`, all requests that are successfully validated by SSL (if applicable for the port settings on the server) are permitted to use the [Certificate Status](#) HTTP API endpoints. This includes requests which do not provide an SSL client certificate.

If the `certificate-authority.certificate-status.authorization-required` setting is `true` or not specified and the `puppet-admin.client-whitelist` setting has one or more entries, only the requests whose Common Name in the SSL client certificate subject matches one of the `client-whitelist` entries are permitted to use the certificate status HTTP API endpoints.

For any other configuration, requests are only permitted to access the certificate status HTTP API endpoints if allowed per the rule definitions in the `trapperkeeper-authorization` "auth.conf" file. See the [auth.conf](#) on page 78 for more information.

In a Future Major Release

The `certificate-status` settings will be ignored completely by Puppet Server. Requests made to the `certificate-status` HTTP API will only be allowed per the `trapperkeeper-authorization` "auth.conf" configuration.

Detecting and Updating

Look at the `certificate-status` settings in your configuration. If `authorization-required` is set to `false` or `client-whitelist` has one or more entries, these settings would be used to authorize access to the certificate status HTTP API instead of `trapperkeeper-authorization`.

If `authorization-required` is set to `true` or is not specified and if the `client-whitelist` was empty, you could just remove the `certificate-authority` section from your configuration. The only behavior that would change in Puppet Server from doing this would be that a warning message would no longer be written to the "puppetserver.log" file at startup.

If `authorization-required` is set to `false`, you would need to create a corresponding rule in the `trapperkeeper-authorization` file which would allow unauthenticated client access to the certificate status API.

For example:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path: "/certificate_status/"
        type: path
        method: [ get, put, delete ]
      }
      allow-unauthenticated: true
      sort-order: 200
      name: "certificate_status"
    },
    {
      match-request: {
        path: "/certificate_statuses/"
        type: path
        method: get
      }
    }
  ]
}
```

```

        allow-unauthenticated: true
        sort-order: 200
        name: "certificate_statuses"
      },
      ...
    ]
  }

```

If `authorization-required` is set to `true` or not set but the `client-whitelist` has one or more custom entries in it, you would need to create a corresponding rule in the `trapperkeeper-authorization "auth.conf"` file which would allow only specific clients access to the certificate status API.

For example, the current certificate status configuration could have:

```

certificate-authority:
  certificate-status: {
    client-whitelist: [ admin1, admin2 ]
  }
}

```

Corresponding `trapperkeeper-authorization` rules could have:

```

authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path: "/certificate_status/"
        type: path
        method: [ get, put, delete ]
      }
      allow: [ admin1, admin2 ]
      sort-order: 200
      name: "certificate_status"
    },
    {
      match-request: {
        path: "/certificate_statuses/"
        type: path
        method: get
      }
      allow: [ admin1, admin2 ]
      sort-order: 200
      name: "certificate_statuses"
    },
    ...
  ]
}

```

After adding the desired rules to the `trapperkeeper-authorization "auth.conf"` file, remove the `certificate-authority` section from the `"puppetserver.conf"` file and restart the `puppetserver` service.

Context

In previous Puppet Server releases, there was no unified mechanism for controlling access to the various endpoints that Puppet Server hosts. Puppet Server used core Puppet `"auth.conf"` to authorize requests handled via Ruby Puppet and custom client whitelists for the CA and Admin endpoints. The custom client whitelists do not provide granular enough control to meet some use cases.

`trapperkeeper-authorization` unifies authorization configuration across all of these endpoints into a single file and provides more granular control.

puppet-admin Settings

Now

If the `puppet-admin.authorization-required` setting is `false`, all requests that are successfully validated by SSL (if applicable for the port settings on the server) are permitted to use the `puppet-admin` HTTP API endpoints. This includes requests which do not provide an SSL client certificate.

If the `puppet-admin.authorization-required` setting is `true` or not specified and the `puppet-admin.client-whitelist` setting has one or more entries, only the requests whose Common Name in the SSL client certificate subject matches one of the `client-whitelist` entries are permitted to use the `puppet-admin` HTTP API endpoints.

For any other configuration, requests are only permitted to access the `puppet-admin` HTTP API endpoints if allowed per the rule definitions in the `trapperkeeper-authorization` "auth.conf" file. See the [auth.conf](#) on page 78 page for more information.

In a Future Major Release

The `puppet-admin` settings will be ignored completely by Puppet Server. Requests made to the `puppet-admin` HTTP API will only be allowed per the `trapperkeeper-authorization` "auth.conf" configuration.

Detecting and Updating

Look at the `puppet-admin` settings in your configuration. If `authorization-required` is set to `false` or `client-whitelist` has one or more entries, these settings would be used to authorize access to the `puppet-admin` HTTP API instead of `trapperkeeper-authorization`.

If `authorization-required` is set to `true` or is not specified and if the `client-whitelist` was empty, you could just remove the `puppet-admin` section from your configuration and restart your `puppetserver` service in order for Puppet Server to start using the `trapperkeeper-authorization` "auth.conf" file. The only behavior that would change in Puppet Server from doing this would be that a warning message would no longer be written to the `puppetserver.log` file.

If `authorization-required` is set to `false`, you would need to create corresponding rules in the `trapperkeeper-authorization` file which would allow unauthenticated client access to the "puppet-admin" API endpoints.

For example:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path: "/puppet-admin-api/v1/environment-cache"
        type: path
        method: delete
      }
      allow-unauthenticated: true
      sort-order: 200
      name: "environment-cache"
    },
    {
      match-request: {
        path: "/puppet-admin-api/v1/jruby-pool"
        type: path
        method: delete
      }
      allow-unauthenticated: true
      sort-order: 200
      name: "jruby-pool"
    },
    ...
  ]
}
```



```
}
```

If `authorization-required` is set to `true` or not set but the `client-whitelist` has one or more custom entries in it, you would need to create corresponding rules in the `trapperkeeper-authorization` "auth.conf" file which would allow only specific clients access to the "puppet-admin" API endpoints.

For example, the current "puppet-admin" configuration could have:

```
puppet-admin: {
  client-whitelist: [ admin1, admin2 ]
}
```

Corresponding `trapperkeeper-authorization` rules could have:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path: "/puppet-admin-api/v1/environment-cache"
        type: path
        method: delete
      }
      allow: [ admin1, admin2 ]
      sort-order: 200
      name: "environment-cache"
    },
    {
      match-request: {
        path: "/puppet-admin-api/v1/jruby-pool"
        type: path
        method: delete
      }
      allow: [ admin1, admin2 ]
      sort-order: 200
      name: "jruby-pool"
    },
    ...
  ]
}
```

After adding the desired rules to the `trapperkeeper-authorization` "auth.conf" file, remove the `puppet-admin` section from the "puppetserver.conf" file and restart the `puppetserver` service.

Context

In previous Puppet Server releases, there was no unified mechanism for controlling access to the various endpoints that Puppet Server hosts. Puppet Server used core Puppet "auth.conf" to authorize requests handled by Ruby Puppet and custom client whitelists for the CA and Admin endpoints. The custom client allowlists do not provide granular enough control to meet some use cases.

`trapperkeeper-authorization` unifies authorization configuration across all of these endpoints into a single file and provides more granular control.

Puppet's "resource_types" API endpoint

Now

The `resource_type` and `resource_types` HTTP APIs were removed in Puppet Server 5.0.

Previously

The [resource_type](#) and [resource_types](#) Puppet HTTP API endpoints return information about classes, defined types, and node definitions.

The [Environment classes](#) on page 192 serves as a replacement for the Puppet resource type API for classes.

Detecting and Updating

If your application calls the `resource_type` or `resource_types` HTTP API endpoints for information about classes, point those calls to the `environment_classes` endpoint. The `environment_classes` endpoint has different features and returns different values than `resource_type`; see the [Environment classes](#) on page 192 for details.

The `environment_classes` endpoint ignores Puppet's Ruby-based authorization methods and configuration in favor of Puppet Server's Trapperkeeper authorization. For more information, see the [Environment classes](#) on page 192 of the environment classes API documentation.

Context

Users often rely on the `resource_types` endpoint for lists of classes and associated parameters in an environment. For such requests, the `resource_types` endpoint is inefficient and can trigger problematic events, such as [manifests being parsed during a catalog request](#).

To fulfill these requests more efficiently and safely, Puppet Server 2.3.0 introduced the narrowly defined `environment_classes` endpoint.

Puppet's node cache terminus

Now

Puppet 5.0 (and by extension, Puppet Server 5.0) no longer writes node YAML files to its cache by default.

Previously

Puppet wrote YAML to its node cache.

Detecting and Updating

To retain the Puppet 4.x behavior, add the [Configuring Puppet Server](#) on page 72 setting `node_cache_terminus = write_only_yaml`. The `write_only_yaml` option is deprecated.

Context

This cache was used in workflows where external tooling needs a list of nodes. PuppetDB is the preferred source of node information.

JRuby's "compat-version" setting

Now

Puppet Server 5.0 removes the `jruby-puppet.compat-version` setting in [puppetserver.conf](#) on page 75, and exits the `puppetserver` service with an error if you start the service with that setting.

Previously

Puppet Server 2.7.x allowed you to set `compat-version` to 1.9 or 2.0 to choose a preferred Ruby interpreter version.

Detecting and Updating

Launching the `puppetserver` service with this setting enabled will cause it to exit with an error message. The error includes information on [Configuring Puppet Server](#) on page 72.

For Ruby language 2.x support in Puppet Server, configure Puppet Server to use JRuby 9k instead of JRuby 1.7.27. See the "Configuring the JRuby Version" section of [Configuring Puppet Server](#) on page 72 for details.

Context

Puppet Server 5.0 updated JRuby v1.7 to v1.7.27, which in turn updated the `jruby-openssl` gem to v0.9.19 and `bouncycastle` libraries to v1.55. JRuby 1.7.27 breaks setting `jruby-puppet.compat-version` to 2.0.

Server 5.0 also added optional, experimental support for JRuby 9k, which includes Ruby 2.x language support.

Compatibility with Puppet agent

Puppet Server 7 is compatible with Puppet agents version 4 and above.

In Platform 7, Puppet Server removed support for the old Puppet 3 HTTP APIs. This means that Puppet 3 agents cannot be used with primary servers running Puppet Server 7.

It also removed the ability to opt in to using the legacy `auth.conf` file. All endpoint access must now be controlled with the HOCON `auth.conf` file. See the [auth.conf docs](#) for details.

Installing Puppet Server

Puppet Server is a required application that runs on the Java Virtual Machine (JVM). It controls the configuration information for one or more managed agent nodes.

Note: If you have any issues with the steps below, submit these to our [bug tracker](#).

Before you begin

Review the supported operating systems and make sure you have a supported version of Java.

Supported operating systems

Puppet provides official packages that install Puppet Server and all of its prerequisites on x86_64 architectures for the following platforms:

- Red Hat Enterprise Linux 6, 7, 8
- Debian 8 (Jessie), 9 (Stretch), 10 (Buster)
- Ubuntu 16.04 (Xenial), 18.04 (Bionic)
- SLES 12 SP1

Note: Unlike Puppet agent, Puppet Server is not supported on MacOS.

Java support

Puppet Server versions are tested against the following versions of Java:

Puppet Server	Java
2.x	7, 8
5.x	8
6.0-6.5	8, 11 (experimental)
6.6 and later	8, 11

Some Java versions may work with other Puppet Server versions, but we do not test or support those cases. Community submitted patches for support greater than Java 11 are welcome. Both Java 8 and 11 are considered long-term support versions and are planned to be supported by upstream maintainers until 2022 or later.

Note: Java 8 runtime packages do not exist in the standard repositories for Debian 8 (Jessie) or Ubuntu 18.04 (Bionic). To install Puppet Server on Jessie, [configure the jessie-backports repository](#). To install Puppet Server on Bionic, enable the [universe repository](#).

Install Puppet Server

Puppet Server is configured to use 2 GB of RAM by default. If you're just testing an installation on a Virtual Machine, this much memory is not necessary. To change the memory allocation, see [Running Puppet Server on a VM](#).

1. [Enable the Puppet package repositories](#), if you haven't already done so.
2. Install the Puppet Server package by running one of the following commands.

Red Hat operating systems:

```
yum install puppetserver
```

Debian and Ubuntu:

```
apt-get install puppetserver
```

There is no `-` in the package name.

Note: If you're upgrading, stop any existing `puppetserver` service by running `service <service_name> stop` or `systemctl stop <service_name>`.

1. Start the Puppet Server service:

```
sudo systemctl start puppetserver
```

1. To check you have installed the Puppet Server package correctly, run the following command to check the version:

```
puppetserver -v
```

What to do next

Now that Puppet Server is installed, move on to these next steps:

1. [Install a Puppet agent](#)
2. [Install PuppetDB](#) (optional) — if you would like to enable extra features, including enhanced queries and reports about your infrastructure.

Running Puppet Server on a VM

By default, Puppet Server is configured to use 2GB of RAM. However, if you want to experiment with Puppet Server on a VM, you can safely allocate as little as 512MB of memory. To change the Puppet Server memory allocation, you can edit the init config file.

- For RHEL or CentOS, open `/etc/sysconfig/puppetserver`
- For Debian or Ubuntu, open `/etc/default/puppetserver`

1. In your settings, update the line:

```
# Modify this if you'd like to change the memory allocation, enable JMX,
etc
JAVA_ARGS="-Xms2g -Xmx2g"
```

Replace 2g with the amount of memory you want to allocate to Puppet Server. For example, to allocate 1GB of memory, use `JAVA_ARGS="-Xms1g -Xmx1g"`; for 512MB, use `JAVA_ARGS="-Xms512m -Xmx512m"`.

For more information about the recommended settings for the JVM, see [Oracle's docs on JVM tuning](#).

2. Restart the `puppetserver` service after making any changes to this file.

Configuring Puppet Server

Configuring Puppet Server

Puppet Server automatically loads the settings in the `main` and `server` sections of the configuration file. If there are duplicates, it prefers the values in the `server` section. Puppet Server honors the following `puppet.conf` settings:

- `allow_duplicate_certs`
- `autosign`
- `cacert`

- cacrl
- cakey
- ca_name
- capub
- ca_ttl
- certdir
- certname
- cert_inventory
- codedir (PE only)
- csrdir
- csr_attributes
- dns_alt_names
- hostcert
- hostcrl
- hostprivkey
- hostpubkey
- keylength
- localcacert
- manage_internal_file_permissions
- privatekeydir
- requestdir
- serial
- signedir
- ssl_client_header
- ssl_client_verify_header
- trusted_oid_mapping_file

However, for some tasks, such as configuring the web server or an external Certificate Authority (CA), Puppet Server has separate configuration files and settings. These files and settings are described below. For more information about differences between Puppet Server and Ruby Puppet's use of `puppet.conf` settings, see [Differing behavior in puppet.conf](#) on page 93.

Configuration Files

Puppet Server's configuration files and settings (with the exception of the [logging config file](#)) are in the `conf.d` directory, located by default at `/etc/puppetlabs/puppetserver/conf.d`. These config files are in the HOCON format, which keeps the basic structure of JSON but is more readable. For more information, see the [HOCON documentation](#).

At startup, Puppet Server reads all the `.conf` files in the `conf.d` directory. You must restart Puppet Server for any changes to those files to take effect. The `conf.d` directory contains the following files and settings:

- [global.conf](#) on page 83
- [webserver.conf](#) on page 82
- [web-routes.conf](#) on page 82
- [puppetserver.conf](#) on page 75
- [auth.conf](#) on page 78
- [ca.conf](#) on page 83

The [product.conf](#) on page 83 file is optional and is not included by default. You can create that file in the `conf.d` directory in order to configure product-related settings, such as automatic update checking and analytics data collection.

Logging

Puppet Server's logging is routed through the JVM [Logback](#) library. The default Logback configuration file is at `/etc/puppetserver/logback.xml` or `/etc/puppetlabs/puppetserver/logback.xml`. You can edit this file to change the logging behavior, or specify a different Logback config file in [global.conf](#).

For more information on the `logback.xml` file, see [logback.xml](#) on page 84 and the [Logback documentation](#). For advanced logging configuration tips, such as configuring Logstash or outputting logs in JSON format, see [Advanced logging configuration](#) on page 85.

For some tips on advanced logging configuration, including information about configuring your system to write logs in a JSON format suitable for sending to logstash or other external logging systems, see the [Advanced logging configuration](#) on page 85 documentation.

HTTP Traffic

Puppet Server logs HTTP traffic in a format similar to Apache, and to a separate file than the main log file. By default, this is located at `/var/log/puppetlabs/puppetserver/puppetserver-access.log`.

By default, the following information is logged for each HTTP request:

- remote host
- remote log name
- remote user
- date of the logging event
- URL requested
- status code of the request
- response content length
- remote IP address
- local port
- elapsed time to serve the request, in milliseconds

The Logback configuration file is at `/etc/puppetlabs/puppetserver/request-logging.xml`. You can edit this file to change the logging behavior. Specify a different Logback configuration file in [webserver.conf](#) with the [access-log-config](#) setting. For more information on configuring the logged data, see [Logback Access Pattern Layout](#).

Authorization

To enable additional logging related to `auth.conf`, edit Puppet Server's `logback.xml` file. By default, only a single message is logged when a request is denied.

To enable a one-time logging of the parsed and transformed `auth.conf` file, add the following to Puppet Server's `logback.xml` file:

```
<logger name="puppetlabs.trapperkeeper.services.authorization.authorization-service" level="DEBUG"/>
```

To enable rule-by-rule logging for each request as it's checked for authorization, add the following to Puppet Server's `logback.xml` file:

```
<logger name="puppetlabs.trapperkeeper.authorization.rules" level="TRACE"/>
```

Service Bootstrapping

Puppet Server is built on top of our open-source Clojure application framework, [Trapperkeeper](#).

One of the features that Trapperkeeper provides is the ability to enable or disable individual services that an application provides. In Puppet Server, you can use this feature to enable or disable the CA service. The CA service is enabled by default, but if you're running a multi-server environment or using an external CA, you might want to disable the CA service on some nodes.

Starting in Puppet Server 2.5.0, the service bootstrap configuration files are in two locations:

- `/etc/puppetlabs/puppetserver/services.d/`: For services that users are expected to manually configure if necessary, such as CA-related settings.
- `/opt/puppetlabs/server/apps/puppetserver/config/services.d/`: For services users *shouldn't* need to configure.

Any files with a `.cfg` extension in either of these locations are combined to form the final set of services Puppet Server will use.

The CA-related configuration settings are set in `/etc/puppetlabs/puppetserver/services.d/ca.cfg`. If services added in future versions have user-configurable settings, the configuration files will also be in this directory. When upgrading Puppet Server 2.5.0 and newer with a package manager, it should not overwrite files already in this directory.

In the `ca.cfg` file, find and modify these lines as directed to enable or disable the service:

```
# To enable the CA service, leave the following line uncommented
puppetlabs.services.ca.certificate-authority-service/certificate-authority-
service
# To disable the CA service, comment out the above line and uncomment the
  line below
#puppetlabs.services.ca.certificate-authority-disabled-service/certificate-
authority-disabled-service
```

Adding Java JARs

Starting with Puppet Server 5.1.0, you can provide Java JARs to be loaded upon Puppet Server's startup.

When launched, Server automatically loads any JARs placed in `/opt/puppetlabs/server/data/puppetserver/jars` into the classpath. JARs placed here will not be modified or removed when upgrading Puppet Server.

Puppet Server configuration files

auth.conf

Puppet Server's `auth.conf` file contains rules for authorizing access to Puppet Server's HTTP API endpoints. For an overview, see [Configuring Puppet Server](#) on page 72.

The rules are defined in a file named `auth.conf`, and Puppet Server applies the settings when a request's endpoint matches a rule.

Note: You can also use the [puppetlabs-puppet_authorization](#) module to manage the new `auth.conf` file's authorization rules in the new HOCON format, and the [puppetlabs-hocon](#) module to use Puppet to manage HOCON-formatted settings in general.

To configure how Puppet Server authenticates requests, use the supported HOCON `auth.conf` file and authorization methods, and see the parameters and rule definitions in the [HOCON Parameters](#) section.

You can find the Puppet Server `auth.conf` file [here](#).

HOCON example

Here is an example authorization section using the HOCON configuration format:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path: "^/my_path/([^/]+)$"
        type: regex
        method: get
      }
    }
  ]
}
```

```

        allow: [ node1, node2, node3, {extensions:{ext_shortname1:
value1, ext_shortname2: value2}} ]
        sort-order: 1
        name: "user-specific my_path"
    },
    {
        match-request: {
            path: "/my_other_path"
            type: path
        }
        allow-unauthenticated: true
        sort-order: 2
        name: "my_other_path"
    },
]
}

```

For a more detailed example of how to use the HOCON configuration format, see [Configuring The Authorization Service](#).

For descriptions of each setting, see the following sections.

HOCON parameters

Use the following parameters when writing or migrating custom authorization rules using the new HOCON format.

version

The `version` parameter is required. In this initial release, the only supported value is 1.

allow-header-cert-info

Note: Puppet Server ignores the setting of the same name in [puppetserver.conf](#) on page 75 in favor of this setting in the `auth.conf` file.

This optional authorization section parameter determines whether to enable [External SSL termination](#) on page 142 on all HTTP endpoints that Puppet Server handles, including the Puppet API, the certificate authority API, and the Puppet Admin API. It also controls how Puppet Server derives the user's identity for authorization purposes. The default value is `false`.

If this setting is `true`, Puppet Server ignores any presented certificate and relies completely on header data to authorize requests.

Warning! This is very insecure; **do not enable this parameter** unless you've secured your network to prevent **any** untrusted access to Puppet Server.

You cannot rename any of the X-Client headers when this setting is enabled, and you must specify identity through the X-Client-Verify, X-Client-DN, and X-Client-Cert headers.

For more information, see [Disable HTTPS for Puppet Server](#) on page 142 in the Puppet Server documentation and [Configuring the Authorization Service](#) in the `trapperkeeper-authorization` documentation.

rules

The required `rules` array of a Puppet Server's HOCON `auth.conf` file determines how Puppet Server responds to a request. Each element is a map of settings pertaining to a rule, and when Puppet Server receives a request, it evaluates that request against each rule looking for a match.

You define each rule by adding parameters to the rule's `match-request` section. A `rules` array can contain as many rules as you need, each with a single `match-request` section.

If a request matches a rule in a `match-request` section, Puppet Server determines whether to allow or deny the request using the `rules` parameters that follow the rule's `match-request` section:

- At least one of:
 - `allow`
 - `allow-unauthenticated`
 - `deny`
- `sort-order` (required)
- `name` (required)

If no rule matches, Puppet Server denies the request by default and returns an HTTP 403/Forbidden response.

match-request

A `match-request` can take the following parameters, some of which are required:

- **path and type (required):** A `match-request` rule must have a `path` parameter, which returns a match when a request's endpoint URL starts with or contains the `path` parameter's value. The parameter can be a literal string or regular expression as defined in the required `type` parameter.

```
# Regular expression to match a path in a URL.
path: "^/puppet/v3/report/([^\/]*)$"
type: regex

# Literal string to match the start of a URL's path.
path: "/puppet/v3/report/"
type: path
```

Note: While the HOCON format doesn't require you to wrap all string values with double quotation marks, some special characters commonly used in regular expressions --- such as `*` --- break HOCON parsing unless the entire value is enclosed in double quotes.

- **method:** If a rule contains the optional `method` parameter, Puppet Server applies that rule only to requests that use its value's listed HTTP methods. This parameter's valid values are `get`, `post`, `put`, `delete`, and `head`, provided either as a single value or array of values.

```
# Use GET and POST.
method: [get, post]

# Use PUT.
method: put
```

Note: While the new HOCON format does not provide a direct equivalent to the [Deprecated features](#) on page 102 `method` parameter's search indirector, you can create the equivalent rule by passing `GET` and `POST` to `method` and specifying endpoint paths using the `path` parameter.

- **query-params:** Use the optional `query-params` setting to provide the list of query parameters. Each entry is a hash of the param name followed by a list of its values.

For example, this rule would match a request URL containing the `environment=production` or `environment=test` query parameters:

```
``` hocon
query-params: {
 environment: [production, test]
}
```
```

`allow`, `allow-unauthenticated`, and `deny`

After each rule's `match-request` section, it must also have an `allow`, `allow-unauthenticated`, or `deny` parameter. (You can set both `allow` and `deny` parameters for a rule, though Puppet Server always prioritizes `deny` over `allow` when a request matches both.)

If a request matches the rule, Puppet Server checks the request's authenticated "name" (see [allow-header-cert-info](#)) against these parameters to determine what to do with the request.

- **allow-unauthenticated:** If this Boolean parameter is set to `true`, Puppet Server allows the request --- even if it can't determine an authenticated name. **This is a potentially insecure configuration** --- be careful when enabling it. A rule with this parameter set to `true` can't also contain the `allow` or `deny` parameters.
- **allow:** This parameter can take a single string value, an array of string values, a single map value with either an `extensions` or `certname` key, or an array of string and map values.

The string values can contain:

- An exact domain name, such as `www.example.com`.
- A glob of names containing a `*` in the first segment, such as `*.example.com` or simply `*`.
- A regular expression surrounded by `/` characters, such as `/example/`.
- A backreference to a regular expression's capture group in the `path` value, if the rule also contains a `type` value of `regex`. For example, if the path for the rule were `"^/example/([^/]+)$"`, you can make a backreference to the first capture group using a value like `$1.domain.org`.

The map values can contain:

- An `extensions` key that specifies an array of matching X.509 extensions. Puppet Server authenticates the request only if each key in the map appears in the request, and each key's value exactly matches.
- A `certname` key equivalent to a bare string.

If the request's authenticated name matches the parameter's value, Puppet Server allows it.

Note: If you are using Puppet Server with the CA disabled, you must use OID values for the extensions. Puppet Server will not be able to resolve [short names](#) in this mode.

- **deny:** This parameter can take the same types of values as the `allow` parameter, but refuses the request if the authenticated name matches --- even if the rule contains an `allow` value that also matches.

Also, in the HOCON Puppet Server authentication method, there is no directly equivalent behavior to the [Deprecated features](#) on page 102 `auth` parameter's `on` value.

sort-order

After each rule's `match-request` section, the required `sort-order` parameter sets the order in which Puppet Server evaluates the rule by prioritizing it on a numeric value between 1 and 399 (to be evaluated before default Puppet rules) or 601 to 998 (to be evaluated after Puppet), with lower-numbered values evaluated first. Puppet Server secondarily sorts rules lexicographically by the name string value's Unicode code points.

```
sort-order: 1
```

name

After each rule's `match-request` section, this required parameter's unique string value identifies the rule to Puppet Server. The name value is also written to server logs and error responses returned to unauthorized clients.

```
name: "my path"
```

Note: If multiple rules have the same name value, Puppet Server will fail to launch.

ca.conf

The `ca.conf` file configures settings for the Puppet Server Certificate Authority (CA) service. For an overview, see [Configuring Puppet Server](#) on page 72.

Signing settings

The `allow-subject-alt-names` setting in the `certificate-authority` section enables you to sign certs with subject alternative names. It is false by default for security reasons, but can be enabled if you need to sign certs with subject alternative names. `puppet cert sign` used to allow this via a flag, but `puppetserver ca sign` requires it to be configured in the config file.

The `allow-authorization-extensions` setting in the `certificate-authority` section enables you to sign certs with authorization extensions. It is false by default for security reasons, but can be enabled if you know you need to sign certs this way. `puppet cert sign` used to allow this via a flag, but `puppetserver ca sign` requires it to be configured in the config file.

Infrastructure CRL settings

Puppet Server is able to create a separate CRL file containing only revocations of Puppet infrastructure nodes. This behavior is turned off by default. To enable it, set `certificate-authority.enable-infra-crl` to `true`.

global.conf

The `global.conf` file contains global configuration settings for Puppet Server. For an overview, see [Configuring Puppet Server](#) on page 72.

You shouldn't typically need to make changes to this file. However, you can change the `logging-config` path for the logback logging configuration file if necessary. For more information about the logback file, see <http://logback.qos.ch/manual/configuration.html>.

Example

```
global: {
  logging-config: /etc/puppetlabs/puppetserver/logback.xml
}
```

logback.xml

Puppet Server's logging is routed through the Java Virtual Machine's [Logback library](#) and configured in an XML file typically named `logback.xml`.

Note: This document covers basic, commonly modified options for Puppet Server logs. Logback is a powerful library with many options. For detailed information on configuring Logback, see the [Logback Configuration Manual](#).

For advanced logging configuration tips specific to Puppet Server, such as configuring Logstash or outputting logs in JSON format, see [Advanced logging configuration](#) on page 85.

Puppet Server logging

By default, Puppet Server logs messages and errors to `/var/log/puppetlabs/puppetserver/puppetserver.log`. The default log level is 'INFO', and Puppet Server sends nothing to `syslog`. You can change Puppet Server's logging behavior by editing `/etc/puppetlabs/puppetserver/logback.xml`, and you can specify a different Logback config file in [global.conf](#).

You can restart the `puppetserver` service for changes to take effect, or enable [configuration scanning](#) to allow changes to be recognized at runtime.

Puppet Server also relies on Logback to manage, rotate, and archive Server log files. Logback archives Server logs when they exceed 10MB, and when the total size of all Server logs exceeds 1GB, it automatically deletes the oldest logs.

Settings level

To modify Puppet Server's logging level, change the `level` attribute of the `root` element. By default, the logging level is set to `info`:

```
<root level="info">
```

Supported logging levels, in order from most to least information logged, are `trace`, `debug`, `info`, `warn`, and `error`. For instance, to enable debug logging for Puppet Server, change `info` to `debug`:

```
<root level="debug">
```

Puppet Server profiling data is included at the debug logging level.

You can also change the logging level for JRuby logging from its defaults of `error` and `info` by setting the `level` attribute of the `jruby` element. For example, to enable debug logging for JRuby, set the attribute to `debug`:

```
<jruby level="debug">
```

Logging location

You can change the file to which Puppet Server writes its logs in the appender section named `F1`. By default, the location is set to `/var/log/puppetlabs/puppetserver/puppetserver.log`:

```
...
  <appender name="F1" class="ch.qos.logback.core.FileAppender">
    <file>/var/log/puppetlabs/puppetserver/puppetserver.log</file>
  ...
```

To change this to `/var/log/puppetserver.log`, modify the contents of the `file` element:

```
<file>/var/log/puppetserver.log</file>
```

The user account that owns the Puppet Server process must have write permissions to the destination path.

scan and scanPeriod

Logback supports noticing and reloading configuration changes without requiring a restart, a feature Logback calls **scanning**. To enable this, set the `scan` and `scanPeriod` attributes in the `<configuration>` element of `logback.xml`:

```
<configuration scan="true" scanPeriod="60 seconds">
```

Due to a [bug in Logback](#), the `scanPeriod` must be set to a value; setting only `scan="true"` will not enable configuration scanning. Scanning is enabled by default in the `logback.xml` configuration packaged with Puppet Server.

Note: The HTTP request log does not currently support the scan feature. Adding the `scan` or `scanPeriod` settings to `request-logging.xml` will have no effect.

HTTP request logging

Puppet Server logs HTTP traffic separately, and this logging is configured in a different Logback configuration file located at `/etc/puppetlabs/puppetserver/request-logging.xml`. To specify a different Logback configuration file, change the `access-log-config` setting in Puppet Server's [webserver.conf](#) on page 82 file.

The HTTP request log uses the same Logback configuration format and settings as the Puppet Server log. It also lets you configure what it logs using patterns, which follow Logback's [PatternLayout format](#).

metrics.conf

The `metrics.conf` file configures Puppet Server's [Monitoring Puppet Server metrics](#) on page 144 and [v2 \(Jolokia\) metrics](#) on page 186.

Settings

All settings in the file are contained in a HOCON `metrics` section.

- `server-id`: A unique identifier to be used as part of the namespace for metrics that this server produces.

- **registries:** A section that contains settings to control which metrics are reported, and how they're reported.
 - **<REGISTRY NAME>:** A section named for a registry that contains its settings. In Puppet Server's case, this section should be `puppetserver`.
 - **metrics-allowed:** An array of metrics to report. See the [Monitoring Puppet Server metrics](#) on page 144 for details about individual metrics.
 - **reporters:** Can contain `jmx` and `graphite` sections with a single Boolean enabled setting to enable or disable each reporter type.
- **reporters:** Configures reporters that distribute metrics to external services or viewers.
 - **graphite:** Contains settings for the Graphite reporter.
 - **host:** A string containing the Graphite server's hostname or IP address.
 - **port:** Contains the Graphite service's port number.
 - **update-interval-seconds:** Sets the interval on which Puppet Server will send metrics to the Graphite server.

Example

Puppet Server ships with a default `metrics.conf` file in Puppet Server's `conf.d` directory, similar to the below example with additional comments.

```
metrics: {
  server-id: localhost
  registries: {
    puppetserver: {
      # specify metrics to allow in addition to those in the default
      list
      #metrics-allowed: ["compiler.compile.production"]

      reporters: {
        jmx: {
          enabled: true
        }
        # enable or disable Graphite metrics reporter
        #graphite: {
          # enabled: true
          #}
      }
    }
  }
}

reporters: {
  #graphite: {
    # # graphite host
    # host: "127.0.0.1"
    # # graphite metrics port
    # port: 2003
    # # how often to send metrics to graphite
    # update-interval-seconds: 5
  }
}
```

product.conf

The `product.conf` file contains settings that determine how Puppet Server interacts with Puppet, Inc., such as automatic update checking and analytics data collection.

Settings

The `product.conf` file doesn't exist in a default Puppet Server installation; to configure its settings, you must create it in Puppet Server's `conf.d` directory (located by default at `/etc/puppetlabs/puppetserver/conf.d`). This file is a [HOCON-formatted](#) configuration file with the following settings:

- Settings in the `product` section configure update checking and analytics data collection:
 - `check-for-updates`: If set to `false`, Puppet Server will not automatically check for updates, and will not send analytics data to Puppet.

If this setting is unspecified (default) or set to `true`, Puppet Server checks for updates upon start or restart, and every 24 hours thereafter, by sending the following data to Puppet:

- Product name
- Puppet Server version
- IP address
- Data collection timestamp

Puppet requests this data as one of the many ways we learn about and work with our community. The more we know about how you use Puppet, the better we can address your needs. No personally identifiable information is collected, and the data we collect is never used or shared outside of Puppet.

Example

```
# Disabling automatic update checks and corresponding analytic data
collection

product: {
  check-for-updates: false
}
```

puppetserver.conf

The `puppetserver.conf` file contains settings for the Puppet Server application. For an overview, see [Configuring Puppet Server](#) on page 72.

Settings

Note: Under most conditions, you won't change the default settings for `server-conf-dir` or `server-code-dir`. However, if you do, also change the equivalent Puppet settings (`confdir` or `codedir`) to ensure that commands like `puppetserver ca` and `puppet module` use the same directories as Puppet Server. You must also specify the non-default `confdir` when running commands, because that setting must be set before Puppet tries to find its config file.

- The `jruby-puppet` settings configure the interpreter.
 - Deprecation Note:** Puppet Server 5.0 removed the `compat-version` setting, which is incompatible with JRuby 1.7.27, and the service won't start if `compat-version` is set. Puppet Server 6.0 uses JRuby 9.1 which supports Ruby 2.3.
 - `ruby-load-path`: The location where Puppet Server expects to find Puppet, Facter, and other components.
 - `gem-home`: The location where JRuby looks for gems. It is also used by the `puppetserver gem` command line tool. If nothing is specified, JRuby uses the Puppet default `/opt/puppetlabs/server/data/puppetserver/jruby-gems`.
 - `gem-path`: The complete "GEM_PATH" for jruby. If set, it should include the `gem-home` directory, as well as any other directories that gems can be loaded from (including the vendored gems directory for gems that ship with puppetserver). The default value is `["/opt/puppetlabs/server/data/puppetserver/`

`jruby-gems", "/opt/puppetlabs/server/data/puppetserver/vendored-jruby-gems", "/opt/puppetlabs/puppet/lib/ruby/vendor_gems"]`.

- `environment-vars`: Optional. A map of environment variables which are made visible to Ruby code running within JRuby, for example, via the Ruby ENV class.

By default, the only environment variables whose values are set into JRuby from the shell are HOME and PATH.

The default value for the GEM_HOME environment variable in JRuby is set from the value provided for the `jruby-puppet.gem-home` key.

Any variable set from the map for the `environment-vars` key overrides these defaults. Avoid overriding HOME, PATH, or GEM_HOME here because these values are already configurable via the shell or `jruby-puppet.gem-home`.

- `server-conf-dir`: Optional. The path to the Puppet [configuration directory](#). The default is `/etc/puppetlabs/puppet`.
- `server-code-dir`: Optional. The path to the Puppet [code directory](#). The default is `/etc/puppetlabs/code`.
- `server-var-dir`: Optional. The path to the Puppet [cache directory](#). The default is `/opt/puppetlabs/server/data/puppetserver`.
- `server-run-dir`: Optional. The path to the run directory, where the service's PID file is stored. The default is `/var/run/puppetlabs/puppetserver`.
- `server-log-dir`: Optional. The path to the log directory. If nothing is specified, it uses the Puppet default `/var/log/puppetlabs/puppetserver`.
- `max-active-instances`: Optional. The maximum number of JRuby instances allowed. The default is 'num-cpus - 1', with a minimum value of 1 and a maximum value of 4. In multithreaded mode, this controls the number of threads allowed to run concurrently through the single JRuby instance.
- `max-requests-per-instance`: Optional. The number of HTTP requests a given JRuby instance will handle in its lifetime. When a JRuby instance reaches this limit, it is flushed from memory and replaced with a fresh one. The default is 0, which disables automatic JRuby flushing.

JRuby flushing can be useful for working around buggy module code that would otherwise cause memory leaks, but it slightly reduces performance whenever a new JRuby instance reloads all of the Puppet Ruby code. If memory leaks from module code are not an issue in your deployment, the default value of 0 performs best.

- `multithreaded`: Optional, false by default. Configures Puppet Server to use a single JRuby instance to process requests that require a JRuby, processing a number of threads up to `max-active-instances` at a time. Reduces the memory footprint of the server by only requiring a single JRuby.
- `max-queued-requests`: Optional. The maximum number of requests that may be queued waiting to borrow a JRuby from the pool. When this limit is exceeded, a 503 "Service Unavailable" response will be returned for all new requests until the queue drops below the limit. If `max-retry-delay` is set to a positive value, then the 503 responses will include a `Retry-After` header indicating a random sleep time after which the client may retry the request. The default is 0, which disables the queue limit.
- `max-retry-delay`: Optional. Sets the upper limit for the random sleep set as a `Retry-After` header on 503 responses returned when `max-queued-requests` is enabled. A value of 0 will cause the `Retry-After` header to be omitted. Default is 1800 seconds which corresponds to the default run interval of the Puppet daemon.
- `borrow-timeout`: Optional. The timeout in milliseconds, when attempting to borrow an instance from the JRuby pool. The default is 1200000.
- `environment-class-cache-enabled`: Optional. Used to control whether the master service maintains a cache in conjunction with the use of the [Environment classes](#) on page 192.

If this setting is set to `true`, Puppet Server maintains the cache. It also returns an Etag header for each GET request to the API. For subsequent GET requests that use the prior Etag value in an `If-None-Match` header,

when the class information available for an environment has not changed, Puppet Server returns an HTTP 304 (Not Modified) response with no body.

If this setting is set to `false` or is not specified, Puppet Server doesn't maintain a cache, an Etag header is not returned for GET requests, and the If-None-Match header for an incoming request is ignored. It therefore parses the latest available code for an environment from disk on every incoming request.

For more information, see the [Environment classes](#) on page 192.

- `compile-mode`: The default value depends on JRuby versions, for 1.7 it is `off`, for 9k it is `jit`. Used to control JRuby's "CompileMode", which may improve performance. A value of `jit` enables JRuby's "just-in-time" compilation of Ruby code. A value of `force` causes JRuby to attempt to pre-compile all Ruby code.
- `profiling-mode`: Optional. Used to enable JRuby's profiler for service startup and set it to one of the supported modes. The default value is `off`, but it can be set to one of `api`, `flat`, `graph`, `html`, `json`, `off`, and `service`. See [ruby-prof](#) for details on what the various modes do.
- `profiler-output-file`: Optional. Used to set the output file to direct JRuby profiler output. Should be a fully qualified path writable by the service user. If not set will default to a random name inside the service working directory.
- The profiler settings configure profiling:
 - `enabled`: If this is set to `true`, Puppet Server enables profiling for the Puppet Ruby code. The default is `true`.
- The versioned-code settings configure commands required to use [static catalogs](#):
 - `code-id-command`: the path to an executable script that Puppet Server invokes to generate a `code_id`. When compiling a static catalog, Puppet Server uses the output of this script as the catalog's `code_id`. The `code_id` associates the catalog with the compile-time version of any [file resources](#) that has a `source` attribute with a `puppet:///URI` value.
 - `code-content-command` contains the path to an executable script that Puppet Server invokes when an agent makes a [Static file content](#) on page 200 API request for the contents of a [file resource](#) that has a `source` attribute with a `puppet:///URI` value.

Note: The Puppet Server process must be able to execute the `code-id-command` and `code-content-command` scripts, and the scripts must return valid content to standard output and an error code of 0. For more information, see the [static catalogs](#) and [Static file content](#) on page 200 documentation.

If you're using static catalogs, you **must** set and use **both** `code-id-command` and `code-content-command`. If only one of those settings are specified, Puppet Server fails to start. If neither setting is specified, Puppet Server defaults to generating catalogs without static features even when an agent requests a static catalog, which the agent will process as a normal catalog.

Examples

```
# Configuration for the JRuby interpreters.

jruby-puppet: {
  ruby-load-path: [/opt/puppetlabs/puppet/lib/ruby/vendor_ruby]
  gem-home: /opt/puppetlabs/server/data/puppetserver/jruby-gems
  gem-path: [/opt/puppetlabs/server/data/puppetserver/jruby-gems, /opt/
puppetlabs/server/data/puppetserver/vendored-jruby-gems]
  environment-vars: { "FOO" : ${FOO}
                      "LANG" : "de_DE.UTF-8" }
  server-conf-dir: /etc/puppetlabs/puppet
  server-code-dir: /etc/puppetlabs/code
  server-var-dir: /opt/puppetlabs/server/data/puppetserver
  server-run-dir: /var/run/puppetlabs/puppetserver
  server-log-dir: /var/log/puppetlabs/puppetserver
  max-active-instances: 1
  max-requests-per-instance: 0
}
```



```

# Settings related to HTTP client requests made by Puppet Server.
# These settings only apply to client connections using the
Puppet::Network::HttpPool
# classes. Client connections using net/http or net/https directly will not
be
# configured with these settings automatically.
http-client: {
  # A list of acceptable protocols for making HTTP requests
  #ssl-protocols: [TLSv1, TLSv1.1, TLSv1.2]

  # A list of acceptable cipher suites for making HTTP requests. For more
  info on available cipher suites, see:
  # http://docs.oracle.com/javase/7/docs/technotes/guides/security/
  SunProviders.html#SunJSSEProvider
  #cipher-suites: [TLS_RSA_WITH_AES_256_CBC_SHA256,
  #                TLS_RSA_WITH_AES_256_CBC_SHA,
  #                TLS_RSA_WITH_AES_128_CBC_SHA256,
  #                TLS_RSA_WITH_AES_128_CBC_SHA]

  # The amount of time, in milliseconds, that an outbound HTTP connection
  # will wait for data to be available before closing the socket. If not
  # defined, defaults to 20 minutes. If 0, the timeout is infinite and if
  # negative, the value is undefined by the application and governed by
the
  # system default behavior.
  #idle-timeout-milliseconds: 1200000

  # The amount of time, in milliseconds, that an outbound HTTP connection
will
  # wait to connect before giving up. Defaults to 2 minutes if not set. If
0,
  # the timeout is infinite and if negative, the value is undefined in the
  # application and governed by the system default behavior.
  #connect-timeout-milliseconds: 120000

  # Whether to enable http-client metrics; defaults to 'true'.
  #metrics-enabled: true
}

# Settings related to profiling the puppet Ruby code.
profiler: {
  enabled: true
}

# Settings related to static catalogs. These paths are examples. There are
no default
# scripts provided with Puppet Server, and no default path for the scripts.
To use static catalog features, you must set
# the paths and provide your own scripts.
versioned-code: {
  code-id-command: /opt/puppetlabs/server/apps/puppetserver/code-id-
command_script.sh
  code-content-command: /opt/puppetlabs/server/apps/puppetserver/code-
content-command_script.sh
}

```

web-routes.conf

The `web-routes.conf` file configures the Puppet Server `web-router-service`, which sets mount points for Puppet Server's web applications. You should not modify these mount points, as Puppet 4 agents rely on Puppet Server mounting them to specific URLs.

For an overview, see [Configuring Puppet Server](#) on page 72. To configure the webserver service, see the [webserver.conf](#) on page 82.

Example

The `web-routes.conf` file looks like this:

```
# Configure the mount points for the web apps.
web-router-service: {
  # These two should not be modified because the Puppet 4 agent expects
  them to
  # be mounted at these specific paths.
  "puppetlabs.services.ca.certificate-authority-service/certificate-
authority-service": "/puppet-ca"
  "puppetlabs.services.master.master-service/master-service": "/puppet"
  "puppetlabs.services.legacy-routes.legacy-routes-service/legacy-routes-
service": ""

  # This controls the mount point for the Puppet administration API.
  "puppetlabs.services.puppet-admin.puppet-admin-service/puppet-admin-
service": "/puppet-admin-api"
}
```

webserver.conf

The `webserver.conf` file configures the Puppet Server webserver service. For an overview, see [Configuring Puppet Server](#) on page 72. To configure the mount points for the Puppet administrative API web applications, see the [web-routes.conf](#) on page 82.

Examples

The `webserver.conf` file looks something like this:

```
# Configure the webserver.
webserver: {
  # Log webserver access to a specific file.
  access-log-config: /etc/puppetlabs/puppetserver/request-logging.xml
  # Require a valid certificate from the client.
  client-auth: need
  # Listen for HTTPS traffic on all available hostnames.
  ssl-host: 0.0.0.0
  # Listen for HTTPS traffic on port 8140.
  ssl-port: 8140
}
```

These are the main values for managing a Puppet Server installation. For further documentation, including a complete list of available settings and values, see [Configuring the Webserver Service](#).

By default, Puppet Server is configured to use the correct Puppet primary server and certificate authority (CA) certificates. If you're using an external CA and providing your own certificates and keys, make sure the SSL-related parameters in `webserver.conf` point to the correct file.

```
webserver: {
  ...
  ssl-cert      : /path/to/server.pem
  ssl-key       : /path/to/server.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
}
```

Migrating to the HOCON `auth.conf` format

Puppet Server 2.2.0 introduced a significant change in how it manages authentication to API endpoints. The older [Puppet `auth.conf`](#) file and whitelist-based authorization method were deprecated in the same release and are now removed in Puppet Server 7. Puppet Server's current `auth.conf` file format (which is different than the old `auth.conf`) is illustrated below in examples.

Use the following examples and methods to convert your authorization rules when upgrading to Puppet Server 2.2.0 and newer. For detailed information about using `auth.conf` rules with Puppet Server, see the [auth.conf](#) on page 78.

Note: To support both Puppet 3 and Puppet 4 agents connecting to Puppet Server, see [Compatibility with Puppet agent](#) on page 107.

Managing rules with Puppet modules

You can reimplement and manage your authorization rules in the new HOCON format and `auth.conf` file by using the [puppetlabs-puppet_authorization](#) Puppet module. See the module's documentation for details.

Converting rules directly

Most of the deprecated authorization rules and settings are available in the new format.

Unavailable rules, settings, or values

The following rules, settings, and values have no direct equivalent in the new HOCON format. If you require them, you must reimplement them differently in the new format.

- **on value of `auth`:** The deprecated `auth` parameter's on value results in a match only when a request provides a client certificate. There is no equivalent behavior in the HOCON format.
- **`allow_ip` or `deny_ip` parameters**
- **`method` parameter's search indirector:** While there is no direct equivalent to the deprecated search indirector, you can create an equivalent HOCON rule. See [below](#) for an example.

Note: Puppet Server considers the state of a request's authentication differently depending on whether the authorization rules use the older Puppet `auth.conf` or newer HOCON formats. An authorization rule that uses the deprecated format evaluates the `auth` parameter as part of rule-matching process. A HOCON authorization rule first determines whether the request matches other parameters of the rule, and then considers the request's authentication state (using the rule's `allow`, `deny`, or `allow-authenticated` values) after a successful match only.

Basic HOCON structure

The HOCON `auth.conf` file has some fundamental structural requirements:

- An [authorization](#) section, which contains:
 - A [version](#) on page 79 setting.
 - A [rules](#) on page 80 array of map values, each representing an authorization rule. Each rule must contain:
 - A [match-request](#) on page 80 section.
 - Each match-request section must contain at least one [path](#) and [type](#).
 - A numeric [sort-order](#) on page 81 value.
 - If the value is between 1 and 399, the rule supersedes Puppet Server's default authorization rules.
 - If the value is between 601 and 998, the rule can be overridden by Puppet Server's default authorization rules.
 - A string [name](#) on page 81 value.
 - At least one of the following:
 - An [allow](#), [allow-unauthenticated](#), and [deny](#) on page 81. The allow or deny values can contain:
 - A single string, representing the request's "name" derived from the Common Name (CN) attribute within an X.509 certificate's Subject Distinguished Name (DN). This string can be an exact name, a glob, or a regular expression.
 - A single map value containing an [extension](#) key.
 - A single map value containing a [certname](#) key.
 - An array of values, including string and map values.
 - An [allow](#), [allow-unauthenticated](#), and [deny](#) on page 81 value, but if present, there cannot also be an allow value.

For an full example of a HOCON `auth.conf` file, see the [HOCON example](#) on page 79.

Converting a simple rule

Let's convert this simple deprecated `auth.conf` authorization rule:

```
path /puppet/v3/environments
method find
allow *
```

We'll start with a skeletal, incomplete HOCON `auth.conf` file:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path:
        type:
      }
      allow:
      sort-order: 1
      name:
    },
  ]
}
```

Next, let's convert each component of the deprecated rule to the new HOCON format.

1. Add the path to the new rule's [match-request](#) on page 80 setting in its match-request section.

```
...
  match-request: {
    path: /puppet/v3/environments
    type:
```

```

    }
    allow:
    sort-order: 1
    name:
  },
  ...

```

2. Next, add its type to the section's [match-request](#) on page 80 setting. Because this is a literal string path, the type is path.

```

...
    match-request: {
      path: /puppet/v3/environments
      type: path
    }
    allow:
    sort-order: 1
    name:
  },
  ...

```

3. The legacy rule has a [method](#) setting, with an indirect value of find that's equivalent to the GET and POST HTTP methods. We can implement these by adding an optional HOCON [match-request](#) on page 80 setting in the rule's match-request section and specifying GET and POST as an array.

```

...
    match-request: {
      path: /puppet/v3/environments
      type: path
      method: [get, post]
    }
    allow:
    sort-order: 1
    name:
  },
  ...

```

4. Next, set the [allow](#) setting. The legacy rule used a * glob, which is also supported in HOCON.

```

...
    match-request: {
      path: /puppet/v3/environments
      type: path
      method: [get, post]
    }
    allow: "*"
    sort-order: 1
    name:
  },
  ...

```

5. Finally, give the rule a unique [name](#) value. Remember that the rule will appear in logs and in the body of error responses to unauthorized clients.

```

...
    match-request: {
      path: /puppet/v3/environments
      type: path
      method: [get, post]
    }
    allow: "*"
    sort-order: 1
    name: "environments"

```

```
    },
    ...

```

Our HOCON `auth.conf` file should now allow all authenticated clients to make GET and POST requests to the `/puppet/v3/environments` endpoint, and should look like this:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path: /puppet/v3/environments
        type: path
        method: [get, post]
      }
      allow: "*"
      sort-order: 1
      name: "environments"
    }
  ]
}
```

Converting more complex rules

Paths set by regular expressions

To convert a regular expression path, enclose it in double quotation marks and slash characters (`/`), and set the `type` to `regex`.

Note: You must escape regular expressions to conform to HOCON standards, which are the same as JSON's and differ from the deprecated format's regular expressions. For instance, the digit-matching regular expression `\d` must be escaped with a second backslash, as `\d`.

Deprecated:

```
path ~ ^/puppet/v3/catalog/([^/]+)$
```

HOCON:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path: "^/puppet/v3/catalog/([^/]+)$"
        type: regex
      }
    }
  ]
}
```

Note: You must escape regular expressions to conform to HOCON standards, which are the same as JSON's and differ from the deprecated format's regular expressions. For instance, the digit-matching regular expression `\d` must be escaped with a second backslash, as `\d`.

Backreferencing works the same way it does in the deprecated format.

Deprecated:

```
path ~ ^/puppet/v3/catalog/([^/]+)$
allow $1
```

HOCON:

```
authorization: {
  version: 1
```

```

rules: [
  {
    match-request: {
      path: "^/puppet/v3/catalog/([^/]+)$"
      type: regex
    }
    allow: "$1"
  }
  ...

```

Allowing unauthenticated requests

To have a rule match any request regardless of its authentication state, including unauthenticated requests, a deprecated rule would assign the any value to the `auth` parameter. In a HOCON rule, set the `allow-unauthenticated` parameter to true. This overrides the `allow` and `deny` parameters and **is an insecure configuration** that should be used with caution.

Deprecated:

```
auth: any
```

HOCON:

```

authorization: {
  version: 1
  rules: [
    {
      match-request: {
        ...
      }
      allow-unauthenticated: true
    }
    ...
  ]
}
...

```

Multiple method indirectors

If a deprecated rule has multiple method indirectors, combine all of the related HTTP methods to the HOCON method array.

Deprecated:

```
method find, save
```

The deprecated `find` indirector corresponds to the GET and POST methods, and the `save` indirector corresponds to the PUT method. In the HOCON format, simply combine these methods in an array.

HOCON:

```

authorization: {
  version: 1
  rules: [
    {
      match-request: {
        ...
        method: [get, post, put]
      }
    }
    ...
  ]
}
...

```

Environment URL parameters

In deprecated rules, the `environment` parameter adds a comma-separated list of query parameters as a suffix to the base URL. HOCON rules allow you to pass them as an array `environment` value inside the `query-params` setting. Rules in both the deprecated and HOCON formats match *any* environment value.

Deprecated:

```
environment: production, test
```

HOCON:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        ...
        query-params: {
          environment: [ production, test ]
        }
      }
    }
  ]
}
...
```

Note: The query-params approach above replaces environment-specific rules for both Puppet 3 and Puppet 4. If you're supporting agents running both Puppet 3 and Puppet 4, see [Compatibility with Puppet agent](#) on page 107 for more information.

Search indirect for method

There's no direct equivalent to the search indirect for the deprecated method setting. Create the equivalent rule by passing GET and POST to method and specifying endpoint paths using the path parameter.

Deprecated:

```
path ~ ^/puppet/v3/file_metadata/user_files/
method search
```

HOCON:

```
authorization: {
  version: 1
  rules: [
    {
      match-request: {
        path: "^/puppet/v3/file_metadata/user_files/"
        type: regex
        method: [get, post]
      }
    }
  ]
}
...
```

Advanced logging configuration

Puppet Server uses the [Logback](#) library to handle all of its logging. Logback configuration settings are stored in the [logback.xml](#) on page 84 file, which is located at `/etc/puppetlabs/puppetserver/logback.xml` by default.

You can configure Logback to log messages in JSON format, which makes it easy to send them to other logging backends, such as Logstash.

Configuring Puppet Server for use with Logstash

There are a few steps necessary to setup your Puppet Server logging for use with Logstash. The first step is to modify your logging configuration so that Puppet Server is logging in a JSON format. After that, you'll configure an external tool to monitor these JSON files and send the data to Logstash (or another remote logging system).

Configuring Puppet Server to log to JSON

Before you configure Puppet Server to log to JSON, consider the following:

- Do you want to configure Puppet Server to *only* log to JSON, instead of the default plain-text logging? Or do you want to have JSON logging *in addition to* the default plain-text logging?
- Do you want to set up JSON logging *only* for the main Puppet Server logs (`puppetserver.log`), or *also* for the HTTP access logs (`puppetserver-access.log`)?
- What kind of log rotation strategy do you want to use for the new JSON log files?

The following examples show how to configure Logback for:

- logging to both JSON and plain-text
- JSON logging both the main logs and the HTTP access logs
- log rotation on the JSON log files

Adjust the example configuration settings to suit your needs.

Note: Puppet Server also relies on Logback to manage, rotate, and archive Server log files. Logback archives Server logs when they exceed 200MB, and when the total size of all Server logs exceeds 1GB, it automatically deletes the oldest logs.

Adding a JSON version of the main Puppet Server logs

Logback writes logs using components called [appenders](#). The example code below uses `RollingFileAppender` to rotate the log files and avoid consuming all of your storage.

1. To configure Puppet Server to log its main logs to a second log file in JSON format, add an appender section like the following example to your `logback.xml` file, at the same level in the XML as existing appenders. The order of the appenders does not matter.

```
<appender name="JSON"
  class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>/var/log/puppetlabs/puppetserver/puppetserver.log.json</file>

  <rollingPolicy
    class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
    <fileNamePattern>/var/log/puppetlabs/puppetserver/
puppetserver.log.json.%d{yyyy-MM-dd}</fileNamePattern>
    <maxHistory>5</maxHistory>
    </rollingPolicy>

    <encoder class="net.logstash.logback.encoder.LogstashEncoder"/>
  </appender>
```

2. Activate the appender by adding an `appender-ref` entry to the `<root>` section of `logback.xml`:

```
<root level="info">
  <appender-ref ref="FILE"/>
  <appender-ref ref="JSON"/>
</root>
```

3. If you decide you want to log *only* the JSON format, comment out the other `appender-ref` entries.

`LogstashEncoder` has many configuration options, including the ability to modify the list of fields that you want to include, or give them different field names. For more information, see the [Logstash Logback Encoder Docs](#).

Adding a JSON version of the Puppet Server HTTP Access logs

To add JSON logging for HTTP requests:

1. Add the following Logback appender section to the `request-logging.xml` file:

```
{% raw %}
<appender name="JSON"
  class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>/var/log/puppetlabs/puppetserver/puppetserver-access.log.json</
file>
```

```

    <rollingPolicy
      class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <fileNamePattern>/var/log/puppetlabs/puppetserver/puppetserver-
access.log.json.%d{yyyy-MM-dd}</fileNamePattern>
        <maxHistory>30</maxHistory>
      </rollingPolicy>

      <encoder
        class="net.logstash.logback.encoder.AccessEventCompositeJsonEncoder">
          <providers>
            <version/>
            <pattern>
              <pattern>
                {
                  "@timestamp": "%date{yyyy-MM-dd'T'HH:mm:ss.SSSXXX}" ,
                  "clientip": "%remoteIP" ,
                  "auth": "%user" ,
                  "verb": "%requestMethod" ,
                  "requestprotocol": "%protocol" ,
                  "rawrequest": "%requestURL" ,
                  "response": "#asLong{%statusCode}" ,
                  "bytes": "#asLong{%bytesSent}" ,
                  "total_service_time": "#asLong{%elapsedTime}" ,
                  "request": "http://%header{Host}%requestURI" ,
                  "referrer": "%header{Referer}" ,
                  "agent": "%header{User-agent}" ,

                  "request.host": "%header{Host}" ,
                  "request.accept": "%header{Accept}" ,
                  "request.accept-encoding": "%header{Accept-
Encoding}" ,

                  "request.connection": "%header{Connection}" ,

                  "puppet.client-verify": "%header{X-Client-Verify}" ,
                  "puppet.client-dn": "%header{X-Client-DN}" ,
                  "puppet.client-cert": "%header{X-Client-Cert}" ,

                  "response.content-type": "%responseHeader{Content-
Type}" ,
                  "response.content-length": "%responseHeader{Content-
Length}" ,

                  "response.server": "%responseHeader{Server}" ,
                  "response.connection": "%responseHeader{Connection}"
                }
              </pattern>
            </pattern>
          </providers>
        </encoder>
      </appender>
    {% endraw %}

```

2. Add a corresponding appender-ref in the configuration section:

```
<appender-ref ref="JSON"/>
```

For more information about options available for the pattern section, see the [Logback Logstash Encoder Docs](#).

Sending the JSON data to Logstash

After configuring Puppet Server to log messages in JSON format, you must also configure it to send the logs to Logstash (or another external logging system). There are several different ways to approach this:

- Configure Logback to send the data to Logstash directly, from within Puppet Server. See the Logstash-Logback encoder docs on how to send the logs by [TCP](#) or [UDP](#). Note that TCP comes with the risk of bottlenecking Puppet Server if your Logstash system is busy, and UDP might silently drop log messages.
- [Filebeat](#) is a tool from Elastic for shipping log data to Logstash.
- [Logstash Forwarder](#) is an earlier tool from Elastic with similar capabilities.

Differing behavior in puppet.conf

Puppet Server honors almost all settings in puppet.conf and should pick them up automatically. For more complete information on puppet.conf settings, see our [Configuration Reference](#) page.

Settings that differ

[autoflush](#)

Puppet Server does not use this setting. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

[bindaddress](#)

Puppet Server does not use this setting. To set the address on which the primary server listens, use either `host` (unencrypted) or `ssl-host` (SSL encrypted) in the [webserver.conf](#) file.

[ca](#)

Puppet Server does not use this setting. Instead, Puppet Server acts as a certificate authority based on the certificate authority service configuration in the `ca.conf` file. See [Service Bootstrapping](#) on page 74 for more details.

[ca_ttl](#)

Puppet Server enforces a max ttl of 50 standard years (up to 1576800000 seconds).

[cacert](#)

If you enable Puppet Server's certificate authority service, it uses the `cacert` setting in puppet.conf to determine the location of the CA certificate for such tasks as generating the CA certificate or using the CA to sign client certificates. This is true regardless of the configuration of the `ssl-` settings in [webserver.conf](#).

[cacrl](#)

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, or `ssl-crl-path` in [webserver.conf](#), Puppet Server uses the file at `ssl-crl-path` as the CRL for authenticating clients via SSL. If at least one of the `ssl-` settings in [webserver.conf](#) is set but `ssl-crl-path` is not set, Puppet Server will *not* use a CRL to validate clients via SSL.

If none of the `ssl-` settings in [webserver.conf](#) are set, Puppet Server uses the CRL file defined for the `hostcrl` setting---and not the file defined for the `cacrl` setting---in puppet.conf. At start time, Puppet Server copies the file for the `cacrl` setting, if one exists, over to the location in the `hostcrl` setting.

Any CRL file updates from the Puppet Server certificate authority---such as revocations performed via the `certificate_status` HTTP endpoint---use the `cacrl` setting in puppet.conf to determine the location of the CRL. This is true regardless of the `ssl-` settings in [webserver.conf](#).

[capass](#)

Puppet Server does not use this setting. Puppet Server's certificate authority does not create a `capass` password file when the CA certificate and key are generated.

[caprivatedir](#)

Puppet Server does not use this setting. Puppet Server's certificate authority does not create this directory.

[daemonize](#)

Puppet Server does not use this setting.

hostcert

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, or `ssl-crl-path` in [webserver.conf](#), Puppet Server presents the file at `ssl-cert` to clients as the server certificate via SSL.

If at least one of the `ssl-` settings in `webserver.conf` is set but `ssl-cert` is not set, Puppet Server gives an error and shuts down at startup. If none of the `ssl-` settings in `webserver.conf` are set, Puppet Server uses the file for the `hostcert` setting in `puppet.conf` as the server certificate during SSL negotiation.

Regardless of the configuration of the `ssl-` "webserver.conf" settings, Puppet Server's certificate authority service, if enabled, uses the `hostcert` "puppet.conf" setting, and not the `ssl-cert` setting, to determine the location of the server host certificate to generate.

hostcrl

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, or `ssl-crl-path` in [webserver.conf](#), Puppet Server uses the file at `ssl-crl-path` as the CRL for authenticating clients via SSL. If at least one of the `ssl-` settings in `webserver.conf` is set but `ssl-crl-path` is not set, Puppet Server will *not* use a CRL to validate clients via SSL.

If none of the `ssl-` settings in `webserver.conf` are set, Puppet Server uses the CRL file defined for the `hostcrl` setting--and not the file defined for the `cacrl` setting--in `puppet.conf`. At start time, Puppet Server copies the file for the `cacrl` setting, if one exists, over to the location in the `hostcrl` setting.

Any CRL file updates from the Puppet Server certificate authority---such as revocations performed via the `certificate_status` HTTP endpoint---use the `cacrl` setting in `puppet.conf` to determine the location of the CRL. This is true regardless of the `ssl-` settings in `webserver.conf`.

hostprivkey

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, or `ssl-crl-path` in [webserver.conf](#), Puppet Server uses the file at `ssl-key` as the server private key during SSL transactions.

If at least one of the `ssl-` settings in `webserver.conf` is set but `ssl-key` is not, Puppet Server gives an error and shuts down at startup. If none of the `ssl-` settings in `webserver.conf` are set, Puppet Server uses the file for the `hostprivkey` setting in `puppet.conf` as the server private key during SSL negotiation.

If you enable the Puppet Server certificate authority service, Puppet Server uses the `hostprivkey` setting in `puppet.conf` to determine the location of the server host private key to generate. This is true regardless of the configuration of the `ssl-` settings in `webserver.conf`.

http_debug

Puppet Server does not use this setting. Debugging for HTTP client code in Puppet Server is controlled through Puppet Server's common logging mechanism. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

keylength

Puppet Server does not currently use this setting. Puppet Server's certificate authority generates 4096-bit keys in conjunction with any SSL certificates that it generates.

localcacert

If you define `ssl-cert`, `ssl-key`, `ssl-ca-cert`, and/or `ssl-crl-path` in [webserver.conf](#), Puppet Server uses the file at `ssl-ca-cert` as the CA cert store for authenticating clients via SSL.

If at least one of the `ssl-` settings in `webserver.conf` is set but `ssl-ca-cert` is not set, Puppet Server gives an error and shuts down at startup. If none of the `ssl-` settings in `webserver.conf` is set, Puppet Server uses the CA file defined for the `localcacert` setting in `puppet.conf` for SSL authentication.

logdir

Puppet Server does not use this setting. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

[masterhttplog](#)

Puppet Server does not use this setting. You can configure a web server access log via the `access-log-config` setting in the [webserver.conf](#) file.

[masterlog](#)

Puppet Server does not use this setting. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

[masterport](#)

Puppet Server does not use this setting. To set the port on which the primary server listens, set the `port` (unencrypted) or `ssl-port` (SSL encrypted) setting in the [webserver.conf](#) file.

[puppetdlog](#)

Puppet Server does not use this setting. For more information on the logging implementation for Puppet Server, see the [Logging](#) on page 74.

[rails_loglevel](#)

Puppet Server does not use this setting.

[railslog](#)

Puppet Server does not use this setting.

[ssl_client_header](#)

Puppet Server honors this setting only if the `allow-header-cert-info` setting in the `server.conf` file is set to 'true'. For more information on this setting, see the documentation on [External SSL termination](#) on page 142.

[ssl_client_verify_header](#)

Puppet Server honors this setting only if the `allow-header-cert-info` setting in the `server.conf` file is set to true. For more information on this setting, see the documentation on [External SSL termination](#) on page 142.

[ssl_server_ca_auth](#)

Puppet Server does not use this setting. It only considers the `ssl-ca-cert` setting from the `webserver.conf` file and the `cacert` setting from the `puppet.conf` file. See [cacert](#) for more information.

[syslogfacility](#)

Puppet Server does not use this setting.

[user](#)

Puppet Server does not use this setting.

HttpPool-Related Server Settings

[configtimeout](#)

Puppet Server does not currently consider this setting when making HTTP requests. This pertains, for example, to any requests that the primary server would make to the `reporturl` for the `http` report processor.

[http_proxy_host](#)

Puppet Server does not currently consider this setting when making HTTP requests. This pertains, for example, to any requests that the primary server would make to the `reporturl` for the `http` report processor.

[http_proxy_port](#)

Puppet Server does not currently consider this setting when making HTTP requests. This pertains, for example, to any requests that the primary server would make to the `reporturl` for the `http` report processor.

Overriding Puppet settings in Puppet Server

Currently, the `jruby-puppet` section of your `puppetserver.conf` file contains five settings (`master-conf-dir`, `master-code-dir`, `master-var-dir`, `master-run-dir`, and `master-log-dir`) that allow you to override settings set in your `puppet.conf` file. On installation, these five settings will be set to the proper default values.

While you are free to change these settings at will, please note that any changes made to the `master-conf-dir` and `master-code-dir` settings absolutely **MUST** be made to the corresponding Puppet settings (`confdir` and `codedir`) as well to ensure that Puppet Server and the Puppet CLI tools (such as `puppetserver ca` and `puppet` module) use the same directories. The `master-conf-dir` and `master-code-dir` settings apply to Puppet Server only, and will be ignored by the ruby code that runs when the Puppet CLI tools are run.

For example, say you have the `codedir` setting left unset in your `puppet.conf` file, and you change the `master-code-dir` setting to `/etc/my-puppet-code-dir`. In this case, Puppet Server will read code from `/etc/my-puppet-code-dir`, but the `puppet` module tool will think that your code is stored in `/etc/puppetlabs/code`.

While it is not as critical to keep `master-var-dir`, `master-run-dir`, and `master-log-dir` in sync with the `vardir`, `rundir`, and `logdir` Puppet settings, please note that this applies to these settings as well.

Also, please note that these configuration differences also apply to the interpolation of the `confdir`, `codedir`, `vardir`, `rundir`, and `logdir` settings in your `puppet.conf` file. So, take the above example, wherein you set `master-code-dir` to `/etc/my-puppet-code-dir`. Because the `basemodulepath` setting is by default `$codedir/modules:/opt/puppetlabs/puppet/modules`, then Puppet Server would use `/etc/my-puppet-code-dir/modules:/opt/puppetlabs/puppet/modules` for the value of the `basemodulepath` setting, whereas the `puppet` module tool would use `/etc/puppetlabs/code/modules:/opt/puppetlabs/puppet/modules` for the value of the `basemodulepath` setting.

Using and extending Puppet Server

Subcommands

We've provided several CLI commands to help with debugging and exploring Puppet Server. Most of the commands are the same ones you would use in a Ruby environment -- such as `gem`, `ruby`, and `irb` -- except they run against Puppet Server's JRuby installation and gems instead of your system Ruby.

The following subcommands are provided:

- `ca`
- `gem`
- `ruby`
- `irb`
- `foreground`

The format for each subcommand is:

```
puppetserver <subcommand> [<args>]
```

When running from source, the format is:

```
lein <subcommand> -c /path/to/puppetserver.conf [--] [<args>]
```

Note that if you are running from source, you need to separate flag arguments (such as `--version` or `-e`) with `--`, as shown above. Otherwise, those arguments will be applied to Leiningen instead of to Puppet Server. This isn't necessary when running from packages (i.e., `puppetserver <subcommand>`).

ca

Available actions

CA subcommand usage: `puppetserver ca <action> [options]` The available actions:

- `clean`: clean files from the CA for certificates
- `generate`: create a new certificate signed by the CA
- `setup`: generate a root and intermediate signing CA for Puppet Server
- `import`: import the CA's key, certs, and CRLs
- `list`: list all certificate requests
- `migrate`: migrate the contents of the CA directory from its current location to `/etc/puppetlabs/puppetserver/ca`. Adds a symlink at the old location for backwards compatibility.
- `revoke`: revoke a given certificate
- `sign`: sign a given certificate
 - Use the `--ttl` flag with `sign` subcommand to send the `ttl` to the CA. The signed certificate's `notAfter` value is the current time plus the `ttl`. The values are valid `puppet.conf` `ttl` values, for example, `1y` = 1 year, `31d` = 31 days.

Most of these subcommands only work if the server is running because they use Puppet Server's API. Exceptions to this requirement are `generate` and `migrate`, which require you to stop the server. You should only run `setup` and `import` before starting the server for the first time.

Syntax:

```
puppetserver ca <action> [options]
```

Most commands require a target to be specified with the `--certname` flag. For example:

```
puppetserver ca sign --certname cert.example.com
```

The target is a comma separated list of names that act on multiple certificates at one time.

You can supply a custom configuration file to all subcommands using the `--config` option. This allows you to point the command at a custom `puppet.conf`, instead of the default one.

Note: These commands are available in Puppet 5, but in order to use them, you must update Puppet Server's `auth.conf` to include a rule allowing the primary server's `certname` to access the `certificate_status` and `certificate_statuses` endpoints. The same applies to upgrading in open source Puppet: if you're upgrading from Puppet 5 to Puppet 6 and are not regenerating your CA, you must allow the primary server's `certname`. See [auth.conf](#) on page 78 for details on how to use `auth.conf`.

Example:

```
{
  # Allow the CA CLI to access the certificate_status endpoint
  match-request: {
    path: "/puppet-ca/v1/certificate_status"
    type: path
    method: [get, put, delete]
  }
  allow: server.example.com
  sort-order: 500
  name: "puppetlabs cert status"
},
```

Signing certs with SANs or auth extensions

With the removal of `puppet cert sign`, it's possible for Puppet Server's CA API to sign certificates with subject alternative names or auth extensions, which was previously completely disallowed. This is disabled by default for security reasons, but you can turn it on by setting `allow-subject-alt-names` or `allow-authorization-extensions` to `true` in the `certificate-authority` section of Puppet Server's config (usually located in `ca.conf`). After these have been configured, you can use `puppetserver ca sign --certname <name>` to sign certificates with these additions.

gem

Installs and manages gems that are isolated from system Ruby and are accessible only to Puppet Server. This is a simple wrapper around the standard Ruby `gem`, so all of the usual arguments and flags should work as expected.

Examples:

```
$ puppetserver gem install pry --no-ri --no-rdoc
```

```
$ lein gem -c /path/to/puppetserver.conf -- install pry --no-ri --no-rdoc
```

If needed, you also can use the `JAVA_ARGS_CLI` environment variable to pass along custom arguments to the Java process that the `gem` command is run within.

Example:

```
$ JAVA_ARGS_CLI=-Xmx8g puppetserver gem install pry --no-ri --no-rdoc
```

If you prefer to have the `JAVA_ARGS_CLI` option persist for multiple command executions, you could set the value in the `/etc/sysconfig/puppetserver` or `/etc/default/puppetserver` file, depending upon your OS distribution:

```
JAVA_ARGS_CLI=-Xmx8g
```

With the value specified in the `sysconfig` or `defaults` file, subsequent commands would use the `JAVA_ARGS_CLI` variable automatically:

```
$ puppetserver gem install pry --no-ri --no-rdoc
// Would run 'gem' with a maximum Java heap of 8g
```

For more information, see [Using Ruby gems](#) on page 138.

ruby

Runs code in Puppet Server's JRuby interpreter. This is a simple wrapper around the standard Ruby `ruby`, so all of the usual arguments and flags should work as expected.

Useful when experimenting with gems installed via `puppetserver gem` and the Puppet and Puppet Server Ruby source code.

Examples:

```
$ puppetserver ruby -e "require 'puppet'; puts Puppet[:certname]"
```

```
$ lein ruby -c /path/to/puppetserver.conf -- -e "require 'puppet'; puts
Puppet[:certname]"
```

If needed, you also can use the `JAVA_ARGS_CLI` environment variable to pass along custom arguments to the Java process that the `ruby` command is run within.

Example:

```
$ JAVA_ARGS_CLI=-Xmx8g puppetserver ruby -e "require 'puppet'; puts
Puppet[:certname]"
```

If you prefer to have the `JAVA_ARGS_CLI` option persist for multiple command executions, you could set the value in the `/etc/sysconfig/puppetserver` or `/etc/default/puppetserver` file, depending upon your OS distribution:

```
JAVA_ARGS_CLI=-Xmx8g
```


With the value specified in the sysconfig or defaults file, subsequent commands would use the `JAVA_ARGS_CLI` variable automatically:

```
$ puppetserver ruby -e "require 'puppet'; puts Puppet[:certname]"
// Would run 'ruby' with a maximum Java heap of 8g
```

irb

Starts an interactive REPL for the JRuby that Puppet Server uses. This is a simple wrapper around the standard Ruby `irb`, so all of the usual arguments and flags should work as expected.

Like the `ruby` subcommand, this is useful for experimenting in an interactive environment with any installed gems (via `puppetserver gem`) and the Puppet and Puppet Server Ruby source code.

Examples:

```
$ puppetserver irb
irb(main):001:0> require 'puppet'
=> true
irb(main):002:0> puts Puppet[:certname]
centos6-64.localdomain
=> nil
```

```
$ lein irb -c /path/to/puppetserver.conf -- --version
irb 0.9.6(09/06/30)
```

If needed, you also can use the `JAVA_ARGS_CLI` environment variable to pass along custom arguments to the Java process that the `irb` command is run within.

Example:

```
$ JAVA_ARGS_CLI=-Xmx8g puppetserver irb
```

If you prefer to have the `JAVA_ARGS_CLI` option persist for multiple command executions, you could set the value in the `/etc/sysconfig/puppetserver` or `/etc/default/puppetserver` file, depending upon your OS distribution:

```
JAVA_ARGS_CLI=-Xmx8g
```

With the value specified in the sysconfig or defaults file, subsequent commands would use the `JAVA_ARGS_CLI` variable automatically:

```
$ puppetserver irb
// Would run 'irb' with a maximum Java heap of 8g
```

foreground

Starts the Puppet Server, but doesn't background it; similar to starting the service and then tailing the log.

Accepts an optional `--debug` argument to raise the logging level to `DEBUG`.

Examples:

```
$ puppetserver foreground --debug
2014-10-25 18:04:22,158 DEBUG [main] [p.t.logging] Debug logging enabled
2014-10-25 18:04:22,160 DEBUG [main] [p.t.bootstrap] Loading bootstrap
config from specified path: '/etc/puppetserver/bootstrap.cfg'
2014-10-25 18:04:26,097 INFO [main] [p.s.j.jruby-puppet-service]
Initializing the JRuby service
2014-10-25 18:04:26,101 INFO [main] [p.t.s.w.jetty9-service] Initializing
web server(s).
```

```
2014-10-25 18:04:26,149 DEBUG [clojure-agent-send-pool-0] [p.s.j.jruby-
puppet-agents] Initializing JRubyPuppet instances with the following
settings:
```

Using Ruby gems

If you have server-side Ruby code in your modules, Puppet Server will run it via JRuby. Generally speaking, this only affects custom parser functions, types, and report processors. For the vast majority of cases this shouldn't pose any problems because JRuby is highly compatible with vanilla Ruby.

Puppet Server will not load gems from user specified `GEM_HOME` and `GEM_PATH` environment variables because `puppetserver` unsets `GEM_PATH` and manages `GEM_HOME`.

Note: Starting with Puppet Server 2.7.1, you can set custom Java arguments for the `puppetserver gem` command via the `JAVA_ARGS_CLI` environment variable, either temporarily on the command line or persistently by adding it to the `sysconfig/default` file. The `JAVA_ARGS_CLI` environment variable also controls the arguments used when running the `puppetserver ruby` and `puppetserver irb` [Subcommands](#) on page 134. See the [Server 2.7.1 release notes](#) for details.

GEM_HOME values

Gems with packaged versions of Puppet Server

The value of `GEM_HOME` when starting the `puppetserver` process as root using a packaged version of `puppetserver` is:

```
/opt/puppetlabs/puppet/cache/jruby-gems
```

This directory does not exist by default.

Gems when running Puppet Server from source

The value of `GEM_HOME` when starting the `puppetserver` process from the project root is:

```
./target/jruby-gems
```

Gems when running Puppet Server spec tests

The value of `GEM_HOME` when starting the `puppetserver JRuby spec` tests using `rake spec` from the project root is:

```
./vendor/test_gems
```

This directory is automatically populated by the `rake spec` task if it does not already exist. The directory may be safely removed and it will be re-populated the next time `rake spec` is run in your working copy.

Installing and removing gems

We isolate the Ruby load paths that are accessible to Puppet Server's JRuby interpreter, so that it doesn't load any gems or other code that you have installed on your system Ruby. If you want Puppet Server to load additional gems, use the Puppet Server-specific `gem` command to install them:

```
$ sudo puppetserver gem install <GEM NAME> --no-document
```

The `puppetserver gem` command is simply a wrapper around the usual Ruby `gem` command, so all of the usual arguments and flags should work as expected. For example, to show your locally installed gems, run:

```
$ puppetserver gem list
```

Or, if you're running from source:

```
$ lein gem -c ~/.puppetserver/puppetserver.conf list
```

The `puppetserver gem` command also respects the running user's `~/.gemrc` file, which you can use to configure upstream sources or proxy settings. For example, consider a `.gemrc` file containing:

```
---
:sources: [ 'https://rubygems-mirror.megacorp.com', 'https://rubygems.org' ]
http_proxy: "http://proxy.megacorp.com:8888"
```

This configures the listed `:sources` as the `puppetserver gem` command's upstream sources, and uses the listed `http_proxy`, which you can confirm:

```
$ puppetserver gem environment | grep proxy
- "http_proxy" => "http://proxy.megacorp.com:8888"
```

As with the rest of Puppet Server's configuration, we recommend managing these settings with Puppet. You can manage Puppet Server's gem dependencies with the package provider shipped in [puppetlabs-puppetserver_gem](#) module.

Note: If you try to load a gem before it's been installed, the agent run will fail with a `LoadError`. If this happens, reload the server after installing the gem to resolve the issue.

Installing gems for use with development:

When running from source, JRuby uses a `GEM_HOME` of `./target/jruby-gems` relative to the current working directory of the process. `lein gem` should be used to install gems into this location using jruby.

NOTE: `./target/jruby-gems` is not used when running the JRuby spec tests, gems are instead automatically installed into and loaded from `./vendor/test_gems`. If you need to install a gem for use both during development and testing make sure the gem is available in both directories.

As an example, the following command installs `pry` locally in the project. Note the use of `--` to pass the following command line arguments to the gem script.

```
$ lein gem --config ~/.puppetserver/puppetserver.conf -- install pry \
--no-document
Fetching: coderay-1.1.0.gem (100%)
Successfully installed coderay-1.1.0
Fetching: slop-3.6.0.gem (100%)
Successfully installed slop-3.6.0
Fetching: method_source-0.8.2.gem (100%)
Successfully installed method_source-0.8.2
Fetching: spoon-0.0.4.gem (100%)
Successfully installed spoon-0.0.4
Fetching: pry-0.10.1-java.gem (100%)
Successfully installed pry-0.10.1-java
5 gems installed
```

With the gem installed into the project tree `pry` can be invoked from inside Ruby code. For more detailed information on `pry` see [pry](#) on page 165.

Gems with Native (C) Extensions

If, in your custom parser functions or report processors, you're using Ruby gems that require native (C) extensions, you won't be able to install these gems under JRuby. In many cases, however, there are drop-in replacements implemented in Java. For example, the popular [Nokogiri](#) gem for processing XML provides a completely compatible Java implementation that's automatically installed if you run `gem install` via JRuby or Puppet Server, so you shouldn't need to change your code at all.

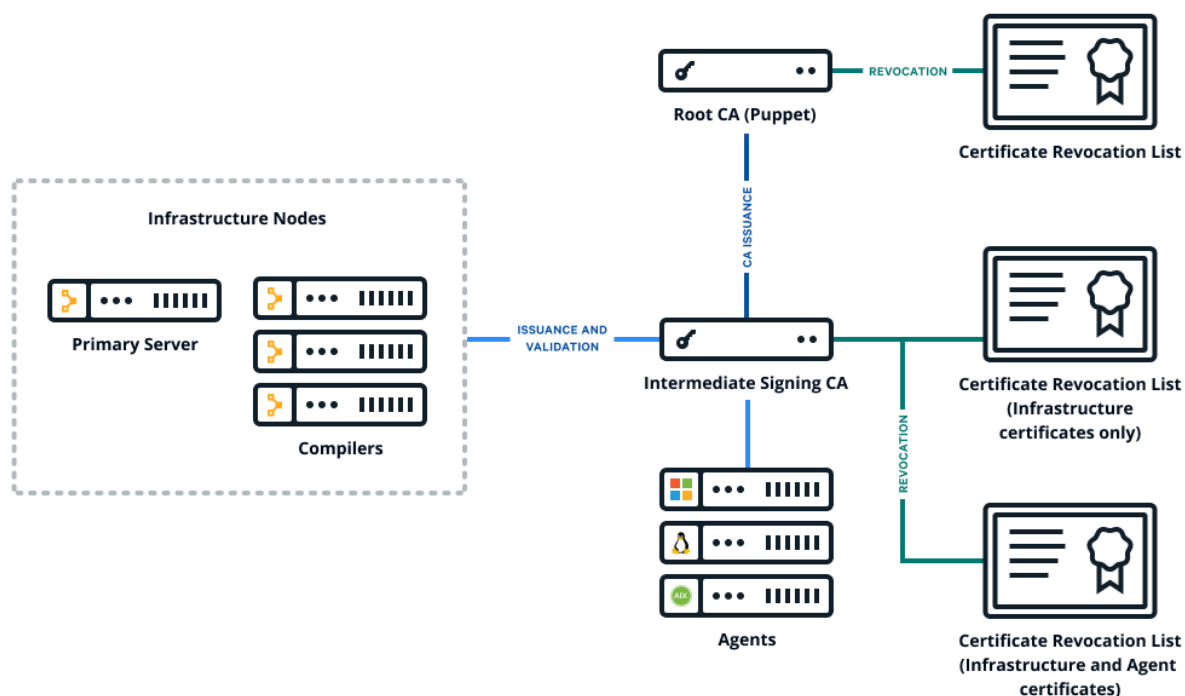
In other cases, there may be a replacement gem available with a slightly different name; e.g., `jdbcmysql` instead of `mysql`. The JRuby wiki [C Extension Alternatives](#) page discusses this issue further.

If you're using a gem that won't run on JRuby and you can't find a suitable replacement, please open a ticket on our [Issue Tracker](#); we're definitely interested in helping provide solutions if there are common gems that are causing trouble for users!

Intermediate CA

Puppet Server supports both a simple CA architecture, with a self-signed root cert that is also used as the CA signing cert; and an intermediate CA architecture, with a self-signed root that issues an intermediate CA cert used for signing incoming certificate requests. The intermediate CA architecture is preferred, because it is more secure and makes regenerating certs easier. To generate a default intermediate CA for Puppet Server, run the `puppetserver ca setup` command before starting your server for the first time.

The following diagram shows the configuration of Puppet's basic certificate infrastructure.



If you have an external certificate authority, you can create a cert chain from it, and use the `puppetserver ca import` subcommand to install the chain on your server. Puppet agents starting with Puppet 6 handle an intermediate CA setup out of the box. No need to copy files around by hand or configure CRL checking. Like `setup`, `import` needs to be run before starting your server for the first time.

Note: The PE installer uses the `puppetserver ca setup` command to create a root cert and an intermediate signing cert for Puppet Server. This means that in PE, the default CA is always an intermediate CA as of PE 2019.0.

Note: If for some reason you cannot use an intermediate CA, in Puppet Server 6 starting the server will generate a non-intermediate CA the same as it always did before the introduction of these commands. However, we don't recommend this, as using an intermediate CA provides more security and easier paths for CA regeneration. It is also the default in PE, and some recommended workflows may rely on it.

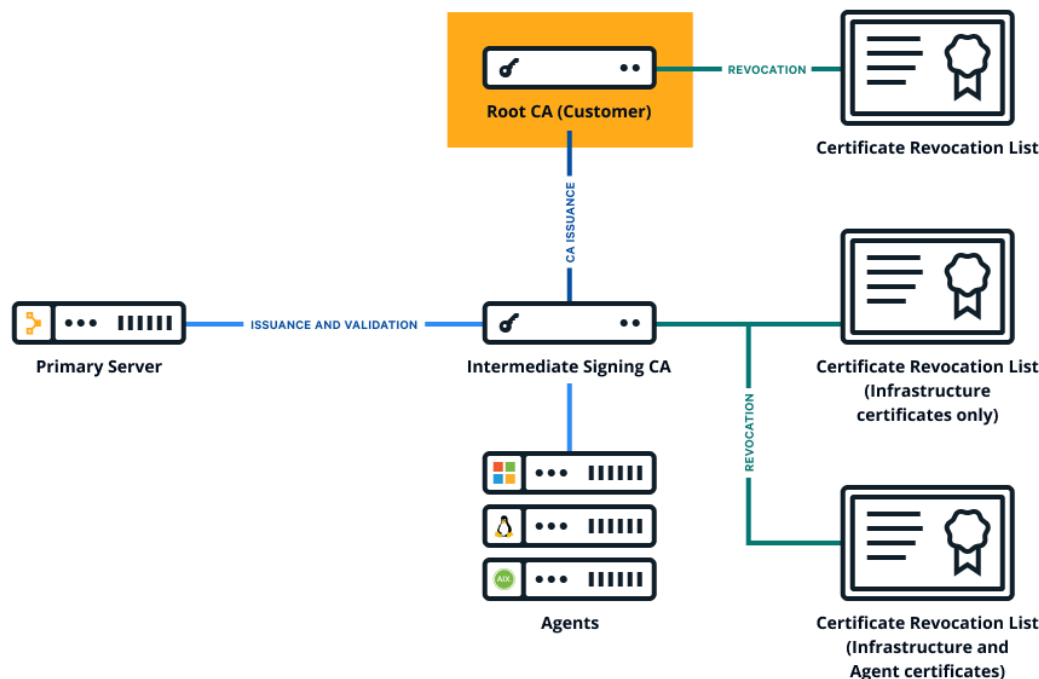
Where to set CA configuration

All CA configuration takes place in Puppet's config file. See the [Puppet Configuration Reference](#) for details.

Set up Puppet as an intermediate CA with an external root

Puppet Server needs to present the full certificate chain to clients so the client can authenticate the server. You construct the certificate chain by concatenating the CA certificates, starting with the new intermediate CA certificate and descending to the root CA certificate.

The following diagram shows the configuration of Puppet's certificate infrastructure with an external root.



To set up Puppet as an intermediate CA with an external root:

1. Collect the PEM-encoded certificates and CRLs for your organization's chain of trust, including the root certificate, any intermediate certificates, and the signing certificate. (The signing certificate might be the root or intermediate certificate.)
2. Create a private RSA key, with no passphrase, for the Puppet CA.
3. Create a PEM-encoded Puppet CA certificate.
 - a. Create a CSR for the Puppet CA.
 - b. Generate the Puppet CA certificate by signing the CSR using your external CA.

Ensure the CA constraint is set to true and the keyIdentifier is composed of the 160-bit SHA-1 hash of the value of the bit string subjectPublicKeyfield. See RFC 5280 section 4.2.1.2 for details.

4. Concatenate all of the certificates into a PEM-encoded certificate bundle, starting with the Puppet CA cert and ending with your root certificate.

```

-----BEGIN CERTIFICATE-----
<Puppet's CA cert>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Org's intermediate CA signing cert>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Org's root CA cert>
-----END CERTIFICATE-----
  
```

- Concatenate all of the CRLs into a PEM-encoded CRL chain, starting with any optional intermediate CA CRLs and ending with your root certificate CRL.

```
-----BEGIN X509 CRL-----
<Puppet's CA CRL>
-----END X509 CRL-----
-----BEGIN X509 CRL-----
<Org's intermediate CA CRL>
-----END X509 CRL-----
-----BEGIN X509 CRL-----
<Org's root CA CRL>
-----END X509 CRL-----
```

- Use the `puppetserver ca import` command to trigger the rest of the CA setup:

```
puppetserver ca import --cert-bundle ca-bundle.pem --crl-chain crls.pem --
private-key puppet_ca_key.pem
```

- optional.

```
openssl x509 -in /etc/puppetlabs/puppet/ssl/ca/signed/<HOSTNAME>.crt
-text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 1 (0x1)
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: CN=intermediate-ca
```

Note: Puppet 5 agents still do not support intermediate CAs. If you must use a Puppet 5 agent with a new (or regenerated) Puppet 6 CA, follow the [instructions](#) for setting up Puppet 5 agents for intermediate CAs.

Infrastructure certificate revocation list (CRL)

The Puppet Server CA can create a CRL that contains only revocations of those nodes that agents are expected to talk to during normal operations, for example, compilers or hosts that agents connect to as part of agent-side functions. Puppet Server CA can distribute that CRL to agents, rather than the CRL it maintains with all node revocations.

To create a smaller CRL, manage the content of the file at `$cadir/infra_inventory.txt`. Provide a newline-separated list of the certnames. When revoked, they are added to the Infra CRL. The certnames must match existing certificates issued and maintained by the Puppet Server CA. Setting the value `certificate-authority.enable-infra-crl` to `true` causes Puppet Server to update both its Full CRL and its Infra CRL with the certs that match those certnames when revoked. When agents first check in, they receive a CRL that includes only the revocations of certnames listed in the `infra_inventory.txt`.

The infrastructure certificate revocation list is disabled by default in open source Puppet. To toggle it, update `enable-infra-crl` in the `certificate-authority` section of `puppetserver.conf`.

This feature is disabled by default because the definition of what constitutes an "infrastructure" node is site-specific and sites with a standard, single primary server configuration have no need for the additional work. After having enabled the feature, if you want to go back, remove the explicit setting and reload Puppet Server to turn the default off; then, when agents first check, they receive the Full CRL as before (including any infrastructure nodes that were revoked while the feature was enabled).

External SSL termination

Use the following steps to configure external SSL termination.

Disable HTTPS for Puppet Server

You'll need to turn off SSL and have Puppet Server use the HTTP protocol instead: remove the `ssl-port` and `ssl-host` settings from the `conf.d/webserver.conf` file and replace them with `port` and `host` settings. See [Configuring the Webserver Service](#) for more information on configuring the web server service.

Allow Client Cert Data From HTTP Headers

When using external SSL termination, Puppet Server expects to receive client certificate information via some HTTP headers.

By default, reading this data from headers is disabled. To allow Puppet Server to recognize it, you'll need to set `allow-header-cert-info: true` in the `authorization` config section of the `/etc/puppetlabs/puppetserver/conf.d/auth.conf` file.

See [Configuring Puppet Server](#) on page 72 for more information on the `puppetserver.conf` and `auth.conf` files.

Note: This assumes the default behavior of Puppet 5 and greater of using Puppet Server's hocon `auth.conf` rather than Puppet's older ini-style `auth.conf`.

WARNING: Setting `allow-header-cert-info` to 'true' puts Puppet Server in an incredibly vulnerable state. Take extra caution to ensure it is **absolutely not reachable** by an untrusted network.

With `allow-header-cert-info` set to 'true', authorization code will use only the client HTTP header values---not an SSL-layer client certificate---to determine the client subject name, authentication status, and trusted facts. This is true even if the web server is hosting an HTTPS connection. This applies to validation of the client via rules in the `auth.conf` file and any [trusted facts](#) extracted from certificate extensions.

If the `client-auth` setting in the `webserver` config block is set to `need` or `want`, the Jetty web server will still validate the client certificate against a certificate authority store, but it will only verify the SSL-layer client certificate---not a certificate in an `X-Client-Cert` header.

Reload Puppet Server

You'll need to reload Puppet Server for the configuration changes to take effect.

Configure SSL Terminating Proxy to Set HTTP Headers

The device that terminates SSL for Puppet Server must extract information from the client's certificate and insert that information into three HTTP headers. See the documentation for your SSL terminator for details.

The headers you'll need to set are `X-Client-Verify`, `X-Client-DN`, and `X-Client-Cert`.

X-Client-Verify

Mandatory. Must be either `SUCCESS` if the certificate was validated, or something else if not. (The convention seems to be to use `NONE` for when a certificate wasn't presented, and `FAILED:reason` for other validation failures.) Puppet Server uses this to authorize requests; only requests with a value of `SUCCESS` will be considered authenticated.

X-Client-DN

Mandatory. Must be the [Subject DN](#) of the agent's certificate, if a certificate was presented. Puppet Server uses this to authorize requests.

X-Client-Cert

Optional. Should contain the client's [PEM-formatted](#) (Base-64) certificate (if a certificate was presented) in a single URI-encoded string. Note that URL encoding is not sufficient; all space characters must be encoded as `%20` and not `+` characters.

Note: Puppet Server only uses the value of this header to extract [trusted facts](#) from extensions in the client certificate. If you aren't using trusted facts, you can choose to reduce the size of the request payload by omitting the `X-Client-Cert` header.

Note: Apache's `mod_proxy` converts line breaks in PEM documents to spaces for some reason, and Puppet Server can't decode the result. We're tracking this issue as [SERVER-217](#).

Server metrics

Monitoring Puppet Server metrics

Puppet Server tracks several advanced performance and health metrics, all of which take advantage of the [v1 metrics](#) on page 185. You can track these metrics using:

- Customizable, networked [Graphite and Grafana instances](#)
- [HTTP Client Metrics](#) on page 151
- [v1 metrics](#) on page 185 endpoints

To visualize Puppet Server metrics, either:

- Export them to a Graphite installation. The [grafanadash](#) module helps you set up a Graphite instance, configure Puppet Server for exporting to it, and visualize the output with Grafana. You can later integrate this with your Graphite installation. For more information, see [Getting started with Graphite](#) below.
- Use the [puppet-metrics-dashboard](#) — this does not go through the Graphite exporting feature. The puppet-metrics-dashboard queries the metrics HTTP API directly and saves the results to disk. It also includes a Docker image of Graphite and Grafana for easy visualization. For more information, see [Puppet Metrics Collection](#).

The puppet-metrics-dashboard is the recommended option for FOSS users, as it is an easier way to save and visualize Puppet Server metrics. The grafanadash is still useful for users exporting to their existing Graphite installation.

Note: The grafanadash and puppet-graphite modules referenced in this document are *not* Puppet-supported modules. They are provided as testing and demonstration purposes *only*.

Getting started with Graphite

[Graphite](#) is a third-party monitoring application that stores real-time metrics and provides customizable ways to view them. Puppet Server can export many metrics to Graphite, and exports a set of metrics by default that is designed to be immediately useful to Puppet administrators.

Note: A Graphite setup is deeply customizable and can report many Puppet Server metrics on demand. However, it requires considerable configuration and additional server resources. To retrieve metrics through HTTP requests, see the [metrics API](#).

To start using Graphite with Puppet Server, you must:

- [Install and configure a Graphite server](#).
- [Enable Puppet Server's Graphite support](#).

[Grafana](#) provides a web-based customizable dashboard that's compatible with Graphite, and the [grafanadash](#) module installs and configures it by default.

Using the grafanadash module to quickly set up a Graphite demo server

The [grafanadash](#) Puppet module quickly installs and configures a basic test instance of [Graphite](#) with the [Grafana](#) extension. When installed on a dedicated Puppet agent, this module provides a quick demonstration of how Graphite and Grafana can consume and display Puppet Server metrics.

WARNING: The grafanadash module is *not* a Puppet-supported module. It is designed for testing and demonstration purposes *only*, and tested against CentOS 6 only.

Also, install this module on a dedicated agent *only*. Do **not** install it on the node running Puppet Server, because the module makes security policy changes that are inappropriate for a Puppet primary server:

- SELinux can cause issues with Graphite and Grafana, so the module temporarily disables SELinux. If you reboot the machine after using the module to install Graphite, you must disable SELinux again and restart the Apache service to use Graphite and Grafana.
- The module disables the iptables firewall and enables cross-origin resource sharing on Apache, which are potential security risks.

Installing the grafanadash Puppet module

Install the grafanadash Puppet module on a *nix agent. The module's grafanadash::dev class installs and configures a Graphite server, the Grafana extension, and a default dashboard.

1. [Install a *nix Puppet agent](#) to serve as the Graphite server.
2. As root on the Puppet agent node, run `puppet module install puppetlabs-grafanadash`.
3. As root on the Puppet agent node, run `puppet apply -e 'include grafanadash::dev'`.

Running Grafana

Grafana runs as a web dashboard, and the `grafanadash` module configures it to use port 10000 by default. To view Puppet metrics in Grafana, you must create a metrics dashboard, or edit and import a JSON-based dashboard that includes Puppet metrics, such as the [sample Grafana dashboard](#) that we provide.

1. In a web browser on a computer that can reach the Puppet agent node running Grafana, navigate to `http://<AGENT'S HOSTNAME>:10000`.

There, you'll see a test screen that indicates whether Grafana can successfully connect to your Graphite server.

If Grafana is configured to use a hostname that the computer on which the browser is running cannot resolve, click **view details** and then the **Requests** tab to determine the hostname Grafana is trying to use. Next, add the IP address and hostname to the computer's `/etc/hosts` file on Linux or OS X, or `C:\Windows\system32\drivers\etc\hosts` file on Windows.

2. Download and edit our [sample Grafana dashboard](#), `sample_metrics_dashboard.json`.
 - a. Open the `sample_metrics_dashboard.json` file in a text editor on the same computer you're using to access Grafana.
 - b. Throughout the file, replace our sample hostname of `server.example.com` with your Puppet Server's hostname. (**Note:** This value **must** be used as the `metrics_server_id` setting, as configured below.)
 - c. Save the file.
3. In the Grafana UI, click **search** (the folder icon), then **Import**, then **Browse**.
4. Navigate to and select the edited JSON file.

This loads a dashboard with nine graphs that display various metrics exported from the Puppet Server to the Graphite server. (For details, see [Using the Grafana dashboard](#).) However, these graphs will remain empty until you enable Puppet Server's Graphite metrics.

Note: If you want to integrate Puppet Server's Grafana exporting with your own infrastructure, use the `grafanadash` module. If you want visualization of metrics, use the `puppetlabs-puppet_metrics_dashboard` module. See [Puppet Metrics Collection](#) for more information.

Enabling Puppet Server's Graphite support

Configure Puppet Server's [metrics.conf](#) on page 116 file to enable and use the Graphite server.

1. Set the enabled parameter to true in `metrics.registries.puppetserver.reporters.graphite`:

```
metrics: {
  server-id: localhost
  registries: {
    puppetserver: {
      ...
      reporters: {
        ...
        # enable or disable Graphite metrics reporter
        graphite: {
          enabled: true
        }
      }
    }
  }
}
```

2. Configure the Graphite host settings in `metrics.reporters.graphite`:
 - **host:** The Graphite host's IP address as a string.
 - **port:** The Graphite host's port number.
 - **update-interval-seconds:** How frequently Puppet Server should send metrics to Graphite.
3. Verify that `metrics.registries.puppetserver.reporters.jmx.enabled` is not set to false. Its default setting is true.

Tip: In the Grafana UI, choose an appropriate time window from the drop-down menu.

Using the sample Grafana dashboard

The [sample Grafana dashboard](#) provides what we think is an interesting starting point. You can click on the title of any graph, and then click **edit** to tweak the graphs as you see fit.

- **Active requests:** This graph serves as a "health check" for the Puppet Server. It shows a flat line that represents the number of CPUs you have in your system, a metric that indicates the total number of HTTP requests actively being processed by the server at any moment in time, and a rolling average of the number of active requests. If the number of requests being processed exceeds the number of CPUs for any significant length of time, your server might be receiving more requests than it can efficiently process.
- **Request durations:** This graph breaks down the average response times for different types of requests made by Puppet agents. This indicates how expensive catalog and report requests are compared to the other types of requests. It also provides a way to see changes in catalog compilation times when you modify your Puppet code. A sharp curve upward for all of the types of requests indicates an overloaded server, and they should trend downward after reducing the load on the server.
- **Request ratios:** This graph shows how many requests of each type that Puppet Server has handled. Under normal circumstances, you should see about the same number of catalog, node, or report requests, because these all happen one time per agent run. The number of file and file metadata requests correlate to how many remote file resources are in the agents' catalogs.
- **Communications with PuppetDB:** This graph tracks the amount of time it takes Puppet Server to send data and requests for common operations to, and receive responses from, PuppetDB.
- **JRubies:** This graph tracks how many JRubies are in use, how many are free, the mean number of free JRubies, and the mean number of requested JRubies.

If the number of free JRubies is often less than one, or the mean number of free JRubies is less than one, Puppet Server is requesting and consuming more JRubies than are available. This overload reduces Puppet Server's performance. While this might simply be a symptom of an under-resourced server, it can also be caused by poorly optimized Puppet code or bottlenecks in the server's communications with PuppetDB if it is in use.

If catalog compilation times have increased but PuppetDB performance remains the same, examine your Puppet code for potentially unoptimized code. If PuppetDB communication times have increased, tune PuppetDB for better performance or allocate more resources to it.

If neither catalog compilation nor PuppetDB communication times are degraded, the Puppet Server process might be under-resourced on your server. If you have available CPU time and memory, [Tuning guide](#) on page 154 to allow it to allocate more JRubies. Otherwise, consider adding additional compilers to distribute the catalog compilation load.

- **JRuby Timers:** This graph tracks several JRuby pool metrics.
 - The borrow time represents the mean amount of time that Puppet Server uses ("borrows") each JRuby from the pool.
 - The wait time represents the total amount of time that Puppet Server waits for a free JRuby instance.
 - The lock held time represents the amount of time that Puppet Server holds a lock on the pool, during which JRubies cannot be borrowed.
 - The lock wait time represents the amount of time that Puppet Server waits to acquire a lock on the pool.

These metrics help identify sources of potential JRuby allocation bottlenecks.

- **Memory Usage:** This graph tracks how much heap and non-heap memory that Puppet Server uses.

- **Compilation:** This graph breaks catalog compilation down into various phases to show how expensive each phase is.

Example Grafana dashboard excerpt

The following example shows only the `targets` parameter of a dashboard to demonstrate the full names of Puppet's exported Graphite metrics (assuming the Puppet Server instance has a domain of `server.example.com`) and a way to add targets directly to an exported Grafana dashboard's JSON content.

```
"panels": [
  {
    "span": 4,
    "editable": true,
    "type": "graphite",
    ...
    "targets": [
      {
        "target": "alias(puppetlabs.server.example.com.num-cpus, 'num
cpus') "
      },
      {
        "target": "alias(puppetlabs.server.example.com.http.active-
requests.count, 'active requests') "
      },
      {
        "target": "alias(puppetlabs.server.example.com.http.active-
histo.mean, 'average') "
      }
    ],
    "aliasColors": {},
    "aliasYAxis": {},
    "title": "Active Requests"
  }
]
```

See the [sample Grafana dashboard](#) for a detailed example of how a Grafana dashboard accesses these exported Graphite metrics.

Available Graphite metrics

The following HTTP and Puppet profiler metrics are available from the Puppet Server and can be added to your metrics reporting. Each metric is prefixed with `puppetlabs.<SERVER-HOSTNAME>`; for instance, the Grafana dashboard file refers to the `num-cpus` metric as `puppetlabs.<SERVER-HOSTNAME>.num-cpus`.

Additionally, metrics might be suffixed by fields, such as `count` or `mean`, that return more specific data points. For instance, the `puppetlabs.<SERVER-HOSTNAME>.compiler.mean` metric returns only the mean length of time it takes Puppet Server to compile a catalog.

To aid with reference, metrics in the list below are segmented into three groups:

- **Statistical metrics:** Metrics that have all eight of these statistical analysis fields, in addition to the top-level metric:
 - `max`: Its maximum measured value.
 - `min`: Its minimum measured value.
 - `mean`: Its mean, or average, value.
 - `stddev`: Its standard deviation from the mean.
 - `count`: An incremental counter.
 - `p50`: The value of its 50th percentile, or median.
 - `p75`: The value of its 75th percentile.
 - `p95`: The value of its 95th percentile.
- **Counters only:** Metrics that only count a value, or only have a `count` field.
- **Other:** Metrics that have unique sets of available fields.

Note: Puppet Server can export many, many metrics -- so many that enabling all of them at large installations can overwhelm Grafana servers. To avoid this, Puppet Server exports only a subset of its available metrics by default. This default set is designed to report the most relevant metrics for administrators monitoring performance and stability.

To add to the default list of exported metrics, see [Modifying Puppet Server's exported metrics](#).

Puppet Server exports each metric in the lists below by default.

Statistical metrics

Compiler metrics

- `puppetlabs.<SERVER-HOSTNAME>.compiler`: The time spent compiling catalogs. This metric represents the sum of the `compiler.compile`, `static_compile`, `find_facts`, and `find_node` fields.
 - `puppetlabs.<SERVER-HOSTNAME>.compiler.compile`: The total time spent compiling dynamic (non-static) catalogs.
- To measure specific nodes and environments, see [Modifying Puppet Server's exported metrics](#).
- `puppetlabs.<SERVER-HOSTNAME>.compiler.find_facts`: The time spent parsing facts.
 - `puppetlabs.<SERVER-HOSTNAME>.compiler.find_node`: The time spent retrieving node data. If the Node Classifier (or another ENC) is configured, this includes the time spent communicating with it.
 - `puppetlabs.<SERVER-HOSTNAME>.compiler.static_compile`: The time spent compiling [static catalogs](#).
 - `puppetlabs.<SERVER-HOSTNAME>.compiler.static_compile_inlining`: The time spent inlining metadata for static catalogs.
 - `puppetlabs.<SERVER-HOSTNAME>.compiler.static_compile_postprocessing`: The time spent post-processing static catalogs.

Function metrics

- `puppetlabs.<SERVER-HOSTNAME>.functions`: The amount of time during catalog compilation spent in function calls. The `functions` metric can also report any of the [statistical metrics](#) fields for a single function by specifying the function name as a field.

For example, to report the mean time spent in a function call during catalog compilation, use `puppetlabs.<SERVER-HOSTNAME>.functions.<FUNCTION-NAME>.mean`.

HTTP metrics

- `puppetlabs.<SERVER-HOSTNAME>.http.active-histo`: A histogram of active HTTP requests over time.
- `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-catalog-/*/-requests`: The time Puppet Server has spent handling catalog requests, including time spent waiting for an available JRuby instance.
- `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-environment_classes-/*/-requests`: The time spent handling requests to the [Environment classes](#) on page 192, which the Node Classifier uses to refresh classes.

- `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-environments-requests`: The time spent handling requests to the [environments API endpoint](#) requests.
- The following metrics measure the time spent handling file-related API endpoints:
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-file_bucket_file-/*/-requests`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-file_content-/*/-requests`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-file_metadata-/*/-requests`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-file_metadatas-/*/-requests`
- `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-node-/*/-requests`: The time spent handling node requests, which are sent to the Node Classifier. A bottleneck here might indicate an issue with the Node Classifier or PuppetDB.
- `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-report-/*/-requests`: The time spent handling report requests. A bottleneck here might indicate an issue with PuppetDB.
- `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-static_file_content-/*/-requests`: The time spent handling requests to the [Static file content](#) on page 200 used by Direct Puppet with file sync.

JRuby metrics

Puppet Server uses an embedded JRuby interpreter to execute Ruby code. By default, JRuby spawns parallel instances known as JRubies to execute Ruby code, which occurs during most Puppet Server activities. When `multithreaded` is set to `true`, a single JRuby is used instead to process a limited number of threads in parallel. For each of these metrics, they refer to JRuby instances by default and JRuby threads in multithreaded mode.

See [Tuning guide](#) on page 154 for details on adjusting JRuby settings.

- `puppetlabs.<SERVER-HOSTNAME>.jruby.borrow-timer`: The time spent with a borrowed JRuby.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.free-jrubies-histo`: A histogram of free JRubies over time. This metric's average value should be greater than 1; if it isn't, [Tuning guide](#) on page 154 or another compile primary server might be needed to keep up with requests.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.lock-held-timer`: The time spent holding the JRuby lock.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.lock-wait-timer`: The time spent waiting to acquire the JRuby lock.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.requested-jrubies-histo`: A histogram of requested JRubies over time. This increases as the number of free JRubies, or the `free-jrubies-histo` metric, decreases, which can suggest that the server's capacity is being depleted.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.wait-timer`: The time spent waiting to borrow a JRuby.

PuppetDB metrics

The following metrics measure the time that Puppet Server spends sending or receiving data from PuppetDB.

- `puppetlabs.<SERVER-HOSTNAME>.puppetdb.catalog.save`
- `puppetlabs.<SERVER-HOSTNAME>.puppetdb.command.submit`
- `puppetlabs.<SERVER-HOSTNAME>.puppetdb.facts.find`
- `puppetlabs.<SERVER-HOSTNAME>.puppetdb.facts.search`
- `puppetlabs.<SERVER-HOSTNAME>.puppetdb.report.process`
- `puppetlabs.<SERVER-HOSTNAME>.puppetdb.resource.search`

Counters only

HTTP metrics

- `puppetlabs.<SERVER-HOSTNAME>.http.active-requests`: The number of active HTTP requests.

- The following counter metrics report the percentage of each HTTP API endpoint's share of total handled HTTP requests.
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-catalog-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-environment-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-environment_classes-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-environments-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-file_bucket_file-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-file_content-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-file_metadata-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-file_metadatas-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-node-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-report-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-resource_type-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-resource_types-/*/-percentage`
 - `puppetlabs.<SERVER-HOSTNAME>.http.puppet-v3-static_file_content-/*/-percentage`
- `puppetlabs.<SERVER-HOSTNAME>.http.total-requests`: The total requests handled by Puppet Server.

JRuby metrics

Note: In multithreaded mode, each of these refers to JRuby threads instead of separate JRuby instances.

- `puppetlabs.<SERVER-HOSTNAME>.jruby.borrow-count`: The number of successfully borrowed JRubies.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.borrow-retry-count`: The number of attempts to borrow a JRuby that must be retried.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.borrow-timeout-count`: The number of attempts to borrow a JRuby that resulted in a timeout.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.request-count`: The number of requested JRubies.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.return-count`: The number of JRubies successfully returned to the pool.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.num-free-jrubies`: The number of free JRuby instances. If this number is often 0, more requests are coming in than the server has available JRuby instances. To alleviate this, increase the number of JRuby instances on the Server or add additional compile servers.
- `puppetlabs.<SERVER-HOSTNAME>.jruby.num-jrubies`: The total number of JRuby instances on the server, governed by the `max-active-instances` setting. See [Tuning guide](#) on page 154 for details.

Other metrics

These metrics measure raw resource availability and capacity.

- `puppetlabs.<SERVER-HOSTNAME>.num-cpus`: The number of available CPUs on the server.
- `puppetlabs.<SERVER-HOSTNAME>.uptime`: The Puppet Server process's uptime.

- Total, heap, and non-heap memory that's committed (committed), initialized (init), and used (used), and the maximum amount of memory that can be used (max).
 - `puppetlabs.<SERVER-HOSTNAME>.memory.total.committed`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.total.init`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.total.used`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.total.max`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.heap.committed`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.heap.init`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.heap.used`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.heap.max`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.non-heap.committed`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.non-heap.init`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.non-heap.used`
 - `puppetlabs.<SERVER-HOSTNAME>.memory.non-heap.max`

For details about HTTP client metrics, which measure performance of Puppet Server's requests to other services, see [HTTP Client Metrics](#) on page 151.

Modifying Puppet Server's exported metrics

In addition to the above default metrics, you can also export metrics measuring specific environments and nodes.

The `metrics.registries.puppetserver.metrics-allowed` parameter in [metrics.conf](#) on page 116 takes an array of strings representing the metrics you want to enable.

Omit the `puppetlabs.<SERVER-HOSTNAME>` prefix and field suffixes (such as `.count` or `.mean`) from metrics when adding them to this class. Instead, suffix the environment or node name as a field to the metric.

For example, to track the compilation time for the production environment, add `compiler.compile.production` to the `metrics-allowed` list. To track only the `my.node.localdomain` node in the production environment, add `compiler.compile.production.my.node.localdomain` to the `metrics-allowed` list.

Optional metrics include:

- `compiler.compile.<ENVIRONMENT>` and `compiler.compile.<ENVIRONMENT>.<NODE-NAME>`, and all statistical fields suffixed to these (such as `compiler.compile.<ENVIRONMENT>.mean`).
- `compiler.compile.evaluate_resources.<RESOURCE>`: Time spent evaluating a specific resource during catalog compilation.

HTTP Client Metrics

HTTP client metrics available in Puppet Server 5 allows users to measure how long it takes for Puppet Server to make requests to and receive responses from other services, such as PuppetDB.

Determining metrics IDs

All of these metrics are of the form `puppetlabs.<SERVER ID>.http-client.experimental.with-metric-id.<METRIC ID>.full-response`.

Note: The `<METRIC ID>` describes what the metric measures. A metric ID is represented in the [Services endpoint](#) on page 189 as an array of strings, and in the metric itself the strings are joined together with periods. For instance, the metric ID of `[puppetdb resource search]` is `puppetdb.resource.search`, so the full metric name would be `puppetlabs.<server-id>.http-client.experimental.with-metric-id.puppetdb.resource.search.full-response`.

You can configure PuppetDB to be a backend for [configuration files](#) (through the `storeconfigs` setting), and you can configure Puppet Server to send reports to an external report processing service. If you configure either of these, then during the course of handling a Puppet agent run, Puppet Server makes several calls to external services to retrieve or store information.

- During handling of a `/puppet/v3/node` request, Puppet Server issues:
 - a `facts find` request to PuppetDB for facts about the node, if they aren't yet cached (typically the first time it requests facts for the node). **Metric ID:** `[puppetdb facts find]`.
- During handling of a `/puppet/v3/catalog` request, Puppet Server issues several requests:
 - a PuppetDB `replace facts` request, to replace the facts for the agent in PuppetDB with the facts it received from the agent. **Metric ID:** `[puppetdb, command, replace_facts]`.
 - a PuppetDB `resource search` request, to search for resources if exported resources are used. **Metric ID:** `[puppetdb, resource, search]`.
 - a PuppetDB `query` request, if the `puppetdb_query` function is used in Puppet code. **Metric ID:** `[puppetdb, query]`.
 - a PuppetDB `replace catalog` request, to replace the catalog for the agent in PuppetDB with the newly compiled catalog. **Metric ID:** `[puppetdb, command, replace_catalog]`.
- During handling of a `/puppet/v3/report` request, Puppet Server issues:
 - a PuppetDB `store report` request, to store the submitted report. **Metric ID:** `[puppetdb command store_report]`.
 - a request to the configured `reports_url` to store the report, if the HTTP report processor is enabled. **Metric ID:** `[puppetdb report http]`.

Configuring

HTTP client metrics are enabled by default, but can be disabled by setting `metrics-enabled` to `false` in the `http-client` section of [puppetserver.conf](#) on page 75.

These metrics also depend on the `server-id` setting in the `metrics` section of `puppetserver.conf`. This defaults to `localhost`, and while `localhost` can collect metrics, change this setting to something unique to avoid metric naming collisions when exporting metrics to an external tool, such as Graphite.

This data is all available via the [Services endpoint](#) on page 189 endpoint, at `https://<SERVER HOSTNAME>:8140/status/v1/services/master?level=debug`. Puppet Server 5.0 adds a `http-client-metrics` keyword in the map. If metrics are not enabled, or if Puppet Server has not issued any requests yet, then this array will be empty, like so: `"http-client-metrics": []`.

In the [sample Grafana dashboard](#), the External HTTP Communications graph visualizes all of these metrics, and the tooltip describes each of them.

Example metrics output

```
"http-client-metrics": [
  {
    "aggregate": 407,
    "count": 1,
    "mean": 407,
    "metric-id": [
      "puppetdb",
      "facts",
      "find"
    ],
    "metric-name": "puppetlabs.localhost.http-client.experimental.with-metric-id.puppetdb.facts.find.full-response"
  },
  {
    "aggregate": 66,
    "count": 1,
    "mean": 66,
    "metric-id": [
      "puppetdb",
      "command",
      "replace_facts"
    ],
  },
]
```



```

    "metric-name": "puppetlabs.localhost.http-client.experimental.with-
metric-id.puppetdb.command.replace_facts.full-response"
  },
  {
    "aggregate": 60,
    "count": 2,
    "mean": 30,
    "metric-id": [
      "puppetdb",
      "resource",
      "search"
    ],
    "metric-name": "puppetlabs.localhost.http-client.experimental.with-
metric-id.puppetdb.resource.search.full-response"
  },
  {
    "aggregate": 53,
    "count": 1,
    "mean": 53,
    "metric-id": [
      "puppetdb",
      "query"
    ],
    "metric-name": "puppetlabs.localhost.http-client.experimental.with-
metric-id.puppetdb.query.full-response"
  },
  {
    "aggregate": 22,
    "count": 1,
    "mean": 22,
    "metric-id": [
      "puppetdb",
      "command",
      "store_report"
    ],
    "metric-name": "puppetlabs.localhost.http-client.experimental.with-
metric-id.puppetdb.command.store_report.full-response"
  },
  {
    "aggregate": 16,
    "count": 1,
    "mean": 16,
    "metric-id": [
      "puppetdb",
      "command",
      "replace_catalog"
    ],
    "metric-name": "puppetlabs.localhost.http-client.experimental.with-
metric-id.puppetdb.command.replace_catalog.full-response"
  },
  {
    "aggregate": 2,
    "count": 1,
    "mean": 2,
    "metric-id": [
      "puppet",
      "report",
      "http"
    ],
    "metric-name": "puppetlabs.localhost.http-client.experimental.with-
metric-id.puppet.report.http.full-response"
  }
],

```

Tuning guide

Puppet Server provides many configuration options that can be used to tune the server for maximum performance and hardware resource utilization. In this guide, we'll highlight some of the most important settings that you can use to get the best performance in your environment.

Puppet Server and JRuby

Before you begin tuning your configuration, it's helpful to have a little bit of context on how Puppet Server uses JRuby to handle incoming HTTP requests from your Puppet agents.

When Puppet Server starts up, it creates a pool of JRuby interpreters to use as workers when it needs need to execute some of the Puppet Ruby code. You can think of these almost as individual Ruby "virtual machines" that are controlled by Puppet Server; it's not entirely dissimilar to the way that Passenger spawns several Ruby processes to hand off work to.

Puppet Server isolates these JRuby instances so that they will only be allowed to handle one request at a time. This ensures that we don't encounter any concurrency issues, because the Ruby code is not thread-safe. When an HTTP request comes in to Puppet Server, and it determines that some Ruby code will need to be executed in order to handle the request, Puppet Server "borrows" a JRuby instance from the pool, uses it to do the work, and then "returns" it to the pool. If there are no JRuby instances available in the pool at the time a request comes in (presumably because all of the JRuby instances are already in use handling other requests), Puppet Server will block the request until one becomes available.

(In the future, this approach will allow us to do some really powerful things such as creating multiple pools of JRubies and isolating each of your Puppet environments to a single pool, to ensure that there is no pollution from one Puppet environment to the next.)

This brings us to the two most important settings that you can use to tune your Puppet Server.

Number of JRubies

The most important setting that you can use to improve the throughput of your Puppet Server installation is the [Configuring Puppet Server](#) on page 72 setting, which you set in [Settings](#) on page 75. The value of this setting is used by Puppet Server to determine how many JRuby instances to create when the server starts up.

From a practical perspective, this setting basically controls how many Puppet agent runs Puppet Server can handle concurrently. The minimum value you can get away with here is 1, and if your installation is small enough that you're unlikely to ever have more than one Puppet agent checking in with the server at exactly the same time, this is totally sufficient.

However, if you specify a value of 1 for this setting, and then you have two Puppet agent runs hitting the server at the same time, the requests being made by the second agent will be effectively blocked until the server has finished handling all of the requests from the first agent. In other words, one of Puppet Server's threads will have "borrowed" the single JRuby instance from the pool to handle the requests from the first agent, and only when those requests are completed will it return the JRuby instance to the pool. At that point, the next thread can "borrow" the JRuby instance to use to handle the requests from the second agent.

Assuming you have more than one CPU core in your machine, this situation means that you won't be getting the maximum possible throughput from your Puppet Server installation. Increasing the value from 1 to 2 would mean that Puppet Server could now use a second CPU core to handle the requests from a second Puppet agent simultaneously.

It follows, then, that the maximum sensible value to use for this setting will be roughly the number of CPU cores you have in your server. Setting the value to something much higher than that won't improve performance, because even if there are extra JRuby instances available in the pool to do work, they won't be able to actually do any work if all of the CPU cores are already busy using JRuby instances to handle incoming agent requests.

(There are some exceptions to this rule. For example, if you have report processors that make a network connection as part of the processing of a report, and if there is a chance that the network operation is slow and will block on I/O for some period of time, then it might make sense to have more JRuby instances than the number of cores. The JVM is smart enough to suspend the thread that is handling those kinds of requests and use the CPUs for other

work, assuming there are still JRuby instances available in the pool. In a case like this you might want to set `max-active-instances` to a value higher than the number of CPUs.)

At this point you may be wondering, "What's the downside to just setting `max-active-instances` to a really high value?" The answer to this question, in a nutshell, is "memory usage". This brings us to the other extremely important setting to consider for Puppet Server.

JVM Heap Size

The JVM's "max heap size" controls the maximum amount of (heap memory that the JVM process is allowed to request from the operating system. You can set this value via the `-Xmx` command-line argument at JVM startup. (In the case of Puppet Server, you'll find this setting in the "defaults" file for Puppet Server for your operating system; this will generally be something like `/etc/sysconfig/puppetserver` or `/etc/defaults/puppetserver`.)

Note: The vast majority of the memory footprint of a JVM process can usually be accounted for by the heap size. However, there is some amount of non-heap memory that will always be used, and for programs that call out to native code at all, there may be a bit more. Generally speaking, the resident memory usage of a JVM process shouldn't exceed the max heap size by more than 256MB or so, but exceeding the max heap size by some amount is normal.

Upgrade note: If you modified the defaults file in Puppet Server 2.4.x or earlier, then lost those modifications or see `Service 'PoolManagerService' not found` warnings after upgrading to Puppet Server 2.5, be aware that the package might have attempted to overwrite the file during the upgrade. See the [Puppet Server 2.5 release notes](#) for details.

If your application's memory usage approaches this value, the JVM will try to get more aggressive with garbage collection to free up memory. In certain situations, you may see increased CPU activity related to this garbage collection. If the JVM is unable to recover enough memory to keep the application running smoothly, you will eventually encounter an `OutOfMemoryError`, and the process will shut down.

For Puppet Server, we also use a JVM argument, `-XX:HeapDumpOnOutOfMemoryError`, to cause the JVM to dump an `.hprof` file to disk. This is basically a memory snapshot at the point in time where the error occurred; it can be loaded into various profiling tools to get a better understanding of where the memory was being used.

(Note that there is another setting, "min heap size", that is controlled via the `-Xms` setting; [Oracle recommends](#) setting this value to the same value that you use for `-Xmx`.)

The most important factor when determining the max heap size for Puppet Server is the value of `max-active-instances`. Each JRuby instance needs to load up a copy of the Puppet Ruby code, and then needs some amount of memory overhead for all of the garbage that gets generated during a Puppet catalog compilation. Also, the memory requirements will vary based on how many Puppet modules you have in your module path, how much Hieradata you have, etc. At this time we estimate that a reasonable ballpark figure is about 512MB of RAM per JRuby instance, but that can vary depending on some characteristics of your Puppet codebase. For example, if you have a really high number of modules or a great deal of Hieradata, you might find that you need more than 512MB per JRuby instance.

You'll also want to allocate a little extra heap to be used by the rest of the things going on in Puppet Server: the web server, etc. So, a good rule of thumb might be `512MB + (max-active-instances * 512MB)`.

We're working on some optimizations for really small installations (for testing, demos, etc.). Puppet Server should run fine with a value of 1 for `max-active-instances` and a heap size of 512MB, and we might be able to improve that further in the future.

Tying Together max-active-instances and Heap Size

We're still gathering data on what the best default settings are, to try to provide an out-of-the-box configuration that works well in most environments. In versions prior to 1.0.8 in the 1.x series (compatible with Puppet 3.x), and prior to 2.1.0 in the 2.x series (compatible with Puppet 4.x), the default value is `num-cpus + 2`. This value will be far too high if you're running on a system with a large number of CPU cores.

As of Puppet Server 1.0.8 and 2.1.0, if you don't provide an explicit value for this setting, we'll default to `num-cpus - 1`, with a minimum value of 1 and a maximum value of 4. The maximum value of 4 is probably too low for production environments with beefy hardware and a high number of Puppet agents checking in, but our current thinking is that it's better to ship with a default setting that is too low and allow you to tune up, than to ship with a

default setting that is too high and causes you to run into `OutOfMemory` errors. In general, it's recommended that you explicitly set this value to something that you think is reasonable in your environment. To encourage this, we log a warning message at startup if you haven't provided an explicit value.

Potential JAVA ARGS settings

If you're working outside of lab environment, increase `ReservedCodeCache` to 512m under normal load. If you're working with 6-12 JRuby instances (or a `max-requests-per-instance` value significantly less than 100k), run with a `ReservedCodeCache` of 1G. Twelve or more JRuby instances in a single server might require 2G or more.

Similar caveats regarding scaling `ReservedCodeCache` might apply if users are managing `MaxMetaspace`.

The `environment_timeout` setting

By default, Puppet does not cache environments, which means the `environment_timeout` setting defaults to 0. This allows you to update code without any extra steps, but it lowers the performance of Puppet Server.

In a production environment, we recommend either:

- Setting the value to `unlimited` and refreshing Puppet Server as part of your code deployment process.
- Setting the value to a number that keeps your most actively used environments cached, but allows testing environments to fall out of the cache and reduce memory usage. For example, a value of 3 minutes (3m).

Applying metrics to improve performance

Puppet Server produces [Monitoring Puppet Server metrics](#) on page 144 that administrators can use to identify performance bottlenecks or capacity issues. Interpreting this data is largely up to you and depends on many factors unique to your installation and usage, but there are some common trends in metrics that you can use to make Puppet Server function better.

Note: This document assumes that you are already familiar with Puppet Server's [Monitoring Puppet Server metrics](#) on page 144, which report on relevant information, and its [Tuning guide](#) on page 154, which provides instructions for modifying relevant settings. To put it another way, this guide attempts to explain questions about "why" Puppet Server performs the way it does for you, while your servers are the "who", Server [Monitoring Puppet Server metrics](#) on page 144 help you track down exactly "what" is affecting performance, and the [Tuning guide](#) on page 154 explains "how" you can improve performance.

If you're using Puppet Enterprise (PE), consult its documentation instead of this guide for PE-specific requirements, settings, and instructions:

- [Large environment installations \(LEI\)](#)
- [Compilers](#)
- [Load balancing](#)
- [High availability](#)

Measuring capacity with JRubies

Puppet Server uses JRuby, which rations server resources in the form of JRuby instances in default mode, and JRuby threads in multithreaded mode. Puppet Server consumes these as it handles requests. A simple way of explaining Puppet Server performance is to remember that your Server infrastructure must be capable of providing enough JRuby instances or threads for the amount of activity it handles. Anything that reduces or limits your server's capacity to produce JRubies also degrades Puppet Server's performance.

Several factors can limit your Server infrastructure's ability to produce JRubies.

Request-handling capacity

Note: These guidelines for interpreting metrics generally apply to both default and multithreaded mode. However, threads are much cheaper in terms of system resources, since they do not need to duplicate all of Puppet's runtime, so you may have more vertical scalability in multithreaded mode.

If your free JRubies are 0 or fewer, your server is receiving more requests for JRubies than it can provide, which means it must queue those requests to wait until resources are available. Puppet Server performs best when the average number of free JRubies is above 1, which means Server always has enough resources to immediately handle incoming requests.

There are two indicators in Puppet Server's metrics that can help you identify a request-handling capacity issue:

- **Average JRuby Wait Time:** This refers to the amount of time Puppet Server has to wait for an available JRuby to become available, and increases when each JRuby is held for a longer period of time, which reduces the overall number of free JRubies and forces new requests to wait longer for available resources.
- **Average JRuby Borrow Time:** This refers to the amount of time that Puppet Server "holds" a JRuby as a resource for a request, and increases because of other factors on the server.

If wait time increases but borrow time stays the same, your Server infrastructure might be serving too many agents. This indicates that Server can easily handle requests but is receiving too many at one time to keep up.

If both wait and borrow times are increasing, something else on your server is causing requests to take longer to process. The longer borrow times suggest that Puppet Server is struggling more than before to process requests, which has a cascading effect on wait times. Correlate borrow time increases with other events whenever possible to isolate what activities might cause them, such as a Puppet code change.

If you are setting up Puppet Server for the first time, start by increasing your Server infrastructure's capacity through additional JRubies (if your server has spare CPU and memory resources) or compilers until you have more than 0 free JRubies, and your average number of free JRubies are at least 1. After your system can handle its request volume, you can start looking into more specific performance improvements.

Adding more JRubies

If you must add JRubies, remember that Puppet Server is tuned by default to use one fewer than your total number of CPUs, with a maximum of 4 CPUs, for the number of available JRubies. You can change this by setting `max-active-instances` in [puppetserver.conf](#) on page 75, under the `jruby-puppet` section. In the default mode, increasing `max-active-instances` creates whole independent JRuby instances. In multithreaded mode, this setting instead controls the number of threads that the single JRuby instance will process concurrently, and therefore has different scaling characteristics. Tuning recommendations for this mode are under development, see [SERVER-2823](#).

When running in the default mode, follow these guidelines for allocating resources when adding JRubies:

Each JRuby also has a certain amount of persistent memory overhead required in order to load both Puppet's Ruby code and your Puppet code. In other words, your available memory sets a baseline limit to how much Puppet code you can process. Catalog compilation can consume more memory, and Puppet Server's total memory usage depends on the number of agents being served, how frequently those agents check in, how many resources are being managed on each agent, and the complexity of the manifests and modules in use.

With the `jruby-puppet.compile-mode` setting in [puppetserver.conf](#) on page 75 set to `off`, a JRuby requires at least 40MB of memory under JRuby 1.7 and at least 60MB under JRuby9k in order to compile a nearly empty catalog. This includes memory for the scripting container, Puppet's Ruby code and additional memory overhead.

For real-world catalogs, you can generally add an absolute minimum of 15MB for each additional JRuby. We calculated this amount by comparing a minimal catalog compilation to compiling a catalog for a [basic role](#) that installs Tomcat and Postgres servers.

Your Puppet-managed infrastructure is probably larger and more complex than that test scenario, and every complication adds more to each additional JRuby's memory requirements. (For instance, we recommend assuming that Puppet Server will use [at least 512MB per JRuby](#) while under load.) You can calculate a similar value unique to your infrastructure by measuring `puppetserver` memory usage during your infrastructure's catalog compilations and comparing it to compiling a minimal catalog for a similar number of nodes.

The `jruby-metrics` section of the [Services endpoint](#) on page 189 endpoint also lists the `requested-instances`, which shows what requests have come in that are waiting to borrow a JRuby instance. This part of the status endpoint lists the lock's status, how many times it has been requested, and how long it has been held for. If it is

currently being held and has been held for a while, you might see requests starting to stack up in the `requested-instances` section.

Adding compilers

If you don't have the additional capacity on your primary server to add more JRubies, you'll want to add another compiler to your Server infrastructure. See [Scaling Puppet Server](#) on page 159.

HTTP request delays

If JRuby metrics appear to be stable, performance issues might originate from lag in server requests, which also have a cascading effect on other metrics. HTTP metrics in the [Services endpoint](#) on page 189, and the requests graph in the [Monitoring Puppet Server metrics](#) on page 144, can help you determine when and where request times have increased.

HTTP metrics include the total time for the server to handle the request, including waiting for a JRuby instance to become available. When JRuby borrow time increases, wait time also increases, so when borrow time for *one* type of request increases, wait times for *all* requests increases.

Catalog compilation, which is graphed on the [sample Grafana dashboard](#), most commonly increases request times, because there are many points of potential failure or delay in a catalog compilation. Several things could cause catalog compilation lengthen JRuby borrow times.

- A Puppet code change, such as a faulty or complex new function. The Grafana dashboard should show if functions start taking significantly longer, and the experimental dashboard and [Services endpoint](#) on page 189 endpoint also list the lengthiest function calls (showing the top 10 and top 40, respectively) based on aggregate execution times.
- Adding many file resources at one time.

In cases like these, there might be more efficient ways to author your Puppet code, you might be extending Puppet to the point where you need to add JRubies or compilers even if you aren't adding more agents.

Slowdowns in PuppetDB can also cause catalog compilations to take more time: if you use exported resources or the `puppetdb_query` function and PuppetDB has a problem, catalog compilation times will increase.

Puppet Server also sends agents' facts and the compiled catalog to PuppetDB during catalog compilation. The [Services endpoint](#) on page 189 for the master service reports metrics for these operations under [HTTP Client Metrics](#) on page 151, and in the Grafana dashboard in the "External HTTP Communications" graph.

Puppet Server also requests facts as HTTP requests while handling a node request, and submits reports via HTTP requests while handling of a report request. If you have an HTTP report processor set up, the Grafana dashboard shows metrics for `Http report processor`, as does the [Services endpoint](#) on page 189 endpoint under `http-client-metrics` in the master service, for metric ID ['puppet', 'report', 'http']. Delays in the report processor are passed on to Puppet Server.

Memory leaks and usage

A memory leak or increased memory pressure can stress Puppet Server's available resources. In this case, the Java VM will spend more time doing garbage collection, causing the GC time and GC CPU % metrics to increase. These metrics are available from the [Services endpoint](#) on page 189 endpoint, as well as in the mbeans metrics available from both the [v1 metrics](#) on page 185 or [v2 \(Jolokia\) metrics](#) on page 186 endpoints.

If you can't identify the source of a memory leak, setting the `max-requests-per-instance` setting in [puppetserver.conf](#) on page 75 to something other than the default of 0 limits the number of requests a JRuby handles during its lifetime and enables automatic JRuby flushing. Enabling this setting reduces overall performance, but if you enable it and no longer see signs of persistent memory leaks, check your module code for inefficiencies or memory-consuming bugs.

Note: In multithreaded mode, the `max-requests-per-instance` setting refers to the sum total number of requests processed by the single JRuby instance, across all of its threads. While that single JRuby is being flushed, all requests will suspend until the instance becomes available again.

Scaling Puppet Server

To scale Puppet Server for many thousands of nodes, you'll need to add more Puppet Server instances dedicated to catalog compilation. These Servers are known as **compilers**, and are simply additional load-balanced Puppet Servers that receive catalog requests from agents and synchronize the results with each other.

If you're using Puppet Enterprise (PE), consult its documentation instead of this guide for PE-specific requirements, settings, and instructions:

- [Large environment installations \(LEI\)](#)
- [Installing compilers](#)
- [High availability](#)
- [Code Manager](#)

Planning your load-balancing strategy

The rest of your configuration depends on how you plan on distributing the agent load. Determine what your deployment will look like before you add any compilers, but **implement load balancing as the last step** only after you have the infrastructure in place to support it.

Using round-robin DNS

Leave all of your agents pointed at the same Puppet Server hostname, then configure your site's DNS to arbitrarily route all requests directed at that hostname to the pool of available servers.

For instance, if all of your agent nodes are configured with `server = puppet.example.com`, configure a DNS name such as:

```
# IP address of server 1:
puppet.example.com. IN A 192.0.2.50
# IP address of server 2:
puppet.example.com. IN A 198.51.100.215
```

For this option, configure your servers with `dns_alt_names` before their certificate request is made.

Using a hardware load balancer

You can also use a hardware load balancer or a load-balancing proxy webserver to redirect requests more intelligently. Depending on your configuration (for instance, SSL using either raw TCP proxying or acting as its own SSL endpoint), you might also need to use other procedures in this document.

Configuring a load balancer depends on the product, and is beyond the scope of this document.

Using DNS SRV Records

You can use DNS SRV records to assign a pool of Puppet Servers for agents to communicate with. This requires a DNS service capable of SRV records, which includes all major DNS software.

Note: This method makes a large number of DNS requests. Request timeouts are completely under the DNS server's control and agents cannot cancel requests early. SRV records don't interact well with static servers set in the config file. Please keep these potential pitfalls in mind when configuring your DNS!

Configure each of your agents with a `srv_domain` instead of a `server` in `puppet.conf`:

```
[main]
use_srv_records = true
srv_domain = example.com
```

Agents will then lookup a SRV record at `_x-puppet._tcp.example.com` when they need to talk to a Puppet server.

```
# Equal-weight load balancing between server-a and server-b:
_x-puppet._tcp.example.com. IN SRV 0 5 8140 server-a.example.com.
_x-puppet._tcp.example.com. IN SRV 0 5 8140 server-b.example.com.
```

You can also implement more complex configurations. For instance, if all devices in site A are configured with a `srv_domain` of `site-a.example.com`, and all nodes in site B are configured to `site-b.example.com`, you can configure them to prefer a server in the local site but fail over to the remote site:

```
# Site A has two servers - server-1 is beefier, give it 75% of the load:
_x-puppet._tcp.site-a.example.com. IN SRV 0 75 8140 server-1.site-
a.example.com.
_x-puppet._tcp.site-a.example.com. IN SRV 0 25 8140 server-2.site-
a.example.com.
_x-puppet._tcp.site-a.example.com. IN SRV 1 5 8140 server.site-
b.example.com.

# For site B, prefer the local server unless it's down, then fail back to
site A
_x-puppet._tcp.site-b.example.com. IN SRV 0 5 8140 server.site-
b.example.com.
_x-puppet._tcp.site-b.example.com. IN SRV 1 75 8140 server-1.site-
a.example.com.
_x-puppet._tcp.site-b.example.com. IN SRV 1 25 8140 server-2.site-
a.example.com.
```

Centralizing the Certificate Authority

Additional Puppet Servers should only share the burden of compiling and serving catalogs, which is why they're typically referred to as "compilers". Any certificate authority functions should be delegated to a single server.

Before you centralize this functionality, ensure that the single server that you want to use as the central CA is reachable at a unique hostname other than (or in addition to) `puppet`. Next, point all agent requests to the centralized CA server, either by configuring each agent or through DNS SRV records.

Directing individual agents to a central CA

On every agent, set the `ca_server` setting in `puppet.conf` (in the `[main]` configuration block) to the hostname of the server acting as the certificate authority. If you have a large number of existing nodes, it is easiest to do this by managing `puppet.conf` with a Puppet module and a template.

Note: Set this setting *before* provisioning new nodes, or they won't be able to complete their initial agent run.

Pointing DNS SRV records at a central CA

If you [use SRV records for agents](#), you can use the `_x-puppet-ca._tcp.$srv_domain` DNS name to point clients to one specific CA server, while the `_x-puppet._tcp.$srv_domain` DNS name handles most of their requests to servers and can point to a set of compilers.

Creating and configuring compilers

To add a compiler to your deployment, begin by [Installing Puppet Server](#) on page 107 on it.

Before running `puppet agent` or starting Puppet Server on the new server:

1. In the compiler's `puppet.conf`, in the `[main]` configuration block, set the `ca_server` setting to the hostname of the server acting as the certificate authority.
2. In the compiler's `webserver.conf` file, add and set the following SSL settings:
 - `ssl-cert`
 - `ssl-key`
 - `ssl-ca-cert`
 - `ssl-crl-path`
3. [Service Bootstrapping](#) on page 74.

If you're using the [individual agent configuration method of CA centralization](#), set `ca_server` in `puppet.conf` to the hostname of your CA server in the `[main]` config block. If an `ssldir` is configured, make sure it's configured in the `[main]` block only.

4. If you're using the [DNS round robin method](#) of agent load balancing, or a [load balancer](#) in TCP proxying mode, provide compilers with certificates using DNS Subject Alternative Names.

Configure `dns_alt_names` in the `[main]` block of `puppet.conf` to cover every DNS name that might be used by an agent to access this server.

```
dns_alt_names = puppet,puppet.example.com,puppet.site-a.example.com
```

If the agent has been run or the server started and already created a certificate, remove it by running `sudo puppet ssl clean`. If an agent has requested a certificate from the server, delete it there to re-issue a new one with the alt names: `puppetserver ca clean server-2.example.com`.

5. Request a new certificate by running `puppet agent --test --waitforcert 10`.
6. Log into the CA server and run `puppetserver ca sign server-2.example.com`.

Centralizing reports, inventory service, and catalog searching (storeconfigs)

If you use an HTTP report processor, point your server and all of your Puppet compilers at the same shared report server in order to see all of your agents' reports.

If you use the inventory service or exported resources, use PuppetDB and point your server and all of your Puppet compilers at a shared PuppetDB instance. A reasonably robust PuppetDB server can handle many Puppet compilers and many thousands of agents.

See the [PuppetDB documentation](#) for instructions on deploying a PuppetDB server, then configure every Puppet compiler to use it. Note that every Puppet primary server and compiler must have its own [allowlist entry](#) if you're using HTTPS certificates for authorization.

Keeping manifests and modules synchronized across compilers

You must ensure that all Puppet compilers have identical copies of your manifests, modules, and [external node classifier](#) data. Examples include:

- Using a version control system such as [r10k](#), Git, Mercurial, or Subversion to manage and sync your manifests, modules, and other data.
- Running an out-of-band `rsync` task via `cron`.
- Configuring `puppet agent` on each compiler to point to a designated model Puppet Server, then use Puppet itself to distribute the modules.

Implementing load distribution

Now that your other compilers are ready, you can implement your [agent load-balancing strategy](#).

Restarting Puppet Server"

Starting in version 2.3.0, you can restart Puppet Server by sending a hangup signal, also known as a [HUP signal](#) or [SIGHUP](#), to the running Puppet Server process. The HUP signal stops Puppet Server and reloads it gracefully, without terminating the JVM process. This is generally *much* faster than completely stopping and restarting the process. This allows you to quickly load changes in Puppet Server, including configuration changes.

There are several ways to send a HUP signal to the Puppet Server process, but the most straightforward is to run the following [kill](#) command:

```
kill -HUP `pgrep -f puppet-server`
```

Starting in version 2.7.0, you can also reload Puppet Server by running the "reload" action via the operating system's service framework. This is analogous to sending a hangup signal but with the benefit of having the "reload" command pause until the server has been completely reloaded, similar to how the "restart" command pauses until the service process has been fully restarted. Advantages to using the "reload" action as opposed to just sending a HUP signal include:

1. Unlike with the HUP signal approach, you do not have to determine the process ID of the `puppetserver` process to be reloaded.

2. When using the HUP signal with an automated script (or Puppet code), it is possible that any additional commands in the script might behave improperly if performed while the server is still reloading. With the "reload" command, though, the server should be up and using its latest configuration before any subsequent script commands are performed.
3. Even if the server fails to reload and shuts down --- for example, due to a configuration error --- the `kill -HUP` command might still return a 0 (success) exit code. With the "reload" command, however, any configuration change which causes the server to shut down will produce a non-0 (failure) exit code. The "reload" command, therefore, would allow you to more reliably determine if the server failed to reload properly.

Use the following commands to perform the "reload" action for Puppet Server.

All current OS distributions:

```
service puppetserver reload
```

OS distributions which use sysvinit-style scripts:

```
/etc/init.d/puppetserver reload
```

OS distributions which use systemd service configurations:

```
systemctl reload puppetserver
```

Note: If you're using Puppet Enterprise (PE), you can reload the server from the command line by running `service pe-puppetserver reload`. However if you need to change a setting, do so in console or with Heira, and then the agent will reload the server when it applies the change. For more information, see [Configuring and tuning Puppet Server](#).

Restarting Puppet Server to pick up changes

There are three ways to trigger your Puppet Server environment to refresh and pick up changes you've made. A request to the [JRuby pool](#) on page 202 is the quickest, but picks up only certain types of changes. A HUP signal or service reload is also quick, and applies additional changes. Other changes require a full Puppet Server restart.

Note: Changes to Puppet Server's [logback.xml](#) on page 84 don't require a server restart. Puppet Server recognizes and applies them automatically, though it can take a minute or so for this to happen. However, you can restart the service to force it to recognize those changes.

Changes applied after a JRuby pool flush, HUP signal, service reload, or full Server restart

- Changes to your `hiera.yaml` file to change your [Hiera](#) configuration.
- [Using Ruby gems](#) on page 138 for Puppet Server by `puppetserver gem`.
- Changes to the Ruby code for Puppet's [core dependencies](#), such as Puppet, Facter, and Hiera.
- Changes to Puppet modules in an [environment](#) where you've enabled [environment caching](#). You can also achieve this by making a request to the [Environment cache](#) on page 202.
- Changes to the CA CRL file. For example, a `puppetserver ca clean`

Changes applied after a HUP signal, service reload, or full Server restart

- Changes to Puppet Server [Configuring Puppet Server](#) on page 72 in its `conf.d` directory.
- Changes to the CA CRL file. For example, a `puppetserver ca clean`

Changes that require a full Server restart

- Changes to JVM arguments, such as [JVM Heap Size](#) on page 155, that are typically configured in your `/etc/sysconfig/puppetserver` or `/etc/default/puppetserver` file.
- Changes to [Service Bootstrapping](#) on page 74 to enable or disable Puppet Server's certificate authority (CA) service.

For these types of changes, you must restart the process by using the operating system's service framework, for example, by using the `systemctl` or `service` commands.

Note: To ensure that the Puppet Server is running in a platform agnostic way, use the `puppet resource service puppetserver ensure=running` command. This command is equivalent to `systemctl start puppetserver` on systems that support it. For more information on the resource command and managing a server's desired state, see [Man Page: puppet resource](#) and [Resource Type: service](#).

Known issues and workarounds

Puppet Server known Issues

For a list of all known issues, visit our [Issue Tracker](#).

Cipher updates in Puppet Server 6.5

Puppet Server 6.5 includes an upgrade to the latest release of Jetty's 9.4 series. With this update, you may see "weak cipher" warnings about ciphers that were previously enabled by default. Puppet Server now defaults to stronger FIPS-compliant ciphers, but you must first remove the weak ciphers.

The ciphers previously enabled by default have not been changed, but are considered weak by the updated standards. Remove the weak ciphers by removing the `cipher-suite` configuration section from the `webserver.conf`. After you remove the `cipher-suite`, Puppet Server uses the FIPS-compliant ciphers instead. This release includes the weak ciphers for backward compatibility only.

The FIPS-compliant cipher suites, which are not considered weak, will be the default in a future version of Puppet. To maintain backwards compatibility, Puppet Server explicitly enables all cipher suites that were available as of Puppet Server 6.0. When you upgrade to Puppet Server 6.5.0, this affects you in two ways:

1. The 6.5 package updates the `webserver.conf` file in Puppet Server's `conf.d` directory.
2. When Puppet Server starts or reloads, Jetty warns about weak cipher suites being enabled.

This update also removes the `so-linger-seconds` configuration setting. This setting is now ignored and a warning is issued if it is set. See Jetty's [so-linger-seconds](#) for removal details.

Note: On some older operating systems, you might see additional warnings that newer cipher suites are unavailable. In this case, manage the contents of the `webserver.cipher-suites` configuration value to be those strong suites that available to you.

Server-side Ruby gems might need to be updated for upgrading from JRuby 1.7

When upgrading from Puppet Server 5 using JRuby 1.7 (9k was optional in those releases), Server-side gems that were installed manually with the `puppetserver gem` command or using the `puppetserver_gem` package provider might need to be updated to work with the newer JRuby. In most cases gems do not have APIs that break when upgrading from the Ruby versions implemented between JRuby 1.7 and JRuby 9k, so there might be no necessary updates. However, two notable exceptions are that the `autosign` gem should be 0.1.3 or later and `yard-doc` must be 0.9 or later.

Potential JAVA ARGS settings

If you're working outside of lab environment, increase `ReservedCodeCache` to 512m under normal load. If you're working with 6-12 JRuby instances (or a `max-requests-per-instance` value significantly less than 100k), run with a `ReservedCodeCache` of 1G. Twelve or more JRuby instances in a single server might require 2G or more.

Similar caveats regarding scaling `ReservedCodeCache` might apply if users are managing `MaxMetaspace`.

`tmp` directory mounted `noexec`

In some cases (especially for RHEL 7 installations) if the `/tmp` directory is mounted as `noexec`, Puppet Server may fail to run correctly, and you may see an error in the Puppet Server logs similar to the following:

```
Nov 12 17:46:12 fqdn.com java[56495]: Failed to load feature test for posix:
can't find user for 0
Nov 12 17:46:12 fqdn.com java[56495]: Cannot run on Microsoft Windows
without the win32-process, win32-dir and win32-service gems: Win32API only
supported on win32
```

```
Nov 12 17:46:12 fqdn.com java[56495]: Puppet::Error: Cannot determine basic
system flavour
```

This is caused by the fact that JRuby contains some embedded files which need to be copied somewhere on the filesystem before they can be executed ([see this JRuby issue](#)). To work around this issue, you can either mount the `/tmp` directory without `noexec`, or you can choose a different directory to use as the temporary directory for the Puppet Server process.

Either way, you'll need to set the permissions of the directory to `1777`. This allows the Puppet Server JRuby process to write a file to `/tmp` and then execute it. If permissions are set incorrectly, you'll get a massive stack trace without much useful information in it.

To use a different temporary directory, you can set the following JVM property:

```
-Djava.io.tmpdir=/some/other/temporary/directory
```

When Puppet Server is installed from packages, add this property to the `JAVA_ARGS` and `JAVA_ARGS_CLI` variables defined in either `/etc/sysconfig/puppetserver` or `/etc/default/puppetserver`, depending on your distribution. Invocations of the `gem`, `ruby`, and `irb` subcommands use the updated `JAVA_ARGS_CLI` on their next invocation. The service will need to be restarted in order to re-read the `JAVA_ARGS` variable.

Puppet Server Fails to Connect to Load-Balanced Servers with Different SSL Certificates

SERVER-207: Intermittent SSL connection failures have been seen when Puppet Server tries to make SSL requests to servers via the same virtual ip address. This has been seen when the servers present different certificates during the SSL handshake.

Developer information

Developer debugging

Because Puppet Server executes both Clojure and Ruby code, approaches to debugging differ depending on which part of the application you're interested in.

Debugging Clojure Code

If you are interested in debugging the web service layer or other parts of the app that are written in Clojure, there are lots of options available. The Clojure REPL is often the most useful tool, as it makes it very easy to interact with individual functions and namespaces.

If you are looking for more traditional debugging capabilities, such as defining breakpoints and stepping through the lines of your source code, there are many options. Just about any Java debugging tool will work to some degree, but Clojure-specific tools such as [CDT](#) and [debug-repl](#) will have better integration with your Clojure source files.

For a more full-featured IDE, [Cursive](#) is a great option. It's built on [IntelliJ IDEA](#), and provides a debug REPL that supports all of the same debugging features that are available in Java; breakpoints, evaluating expressions in the local scope when stopped at a breakpoint, visual navigation of the call stack across threads, etc.

Debugging Ruby Code

Debugging the Ruby code running in Puppet Server can be a bit trickier, because Java and Clojure debugging tools will only take you into the JRuby interpreter source code, not into the Ruby code that it is processing. So, if you wish to debug the Ruby code directly, you'll need to install gems and take advantage of their capabilities (not unlike how you would debug Ruby code in the MRI interpreter).

For more info on installing gems for Puppet Server, see [Using Ruby gems](#) on page 138.

Ruby REPL incompatible with Lein REPL

Please note that a REPL running in Ruby is incompatible with `lein repl` because JRuby will not receive data from standard input when running inside of `lein repl`. To use a ruby REPL during development run `puppetserver` from source with `lein run` rather than `lein repl`:

```
$ lein run --config ~/.puppetserver/puppetserver.conf
```

The `lein run` command will start the server in the foreground as normal. `pry` or `ruby-debug` will display an input prompt when the relevant statement is reached. Expect to see the normal `lein run` output and then the Ruby REPL will present itself as compared to `lein repl` which presents a prompt early in the process lifecycle. In this way the "ruby repl" is more of a breakpoint than a REPL in the Clojure sense.

ruby-debug

Installation

There are many gems available that provide various ways of debugging Ruby code depending on what version of Ruby and which Ruby interpreter you're running. One of the most common gems is `ruby-debug`, and there is a JRuby-compatible version available. To install it for use in Puppet Server, run:

```
$ sudo puppetserver gem install ruby-debug
```

Or, if you're running `puppetserver` from source:

```
$ lein gem -c /path/to/puppetserver.conf install ruby-debug
```

Usage

After installing the gem, you can trigger the debugger by adding a line like this to any of the Ruby code that is run in Puppet Server (including the Puppet Ruby code):

```
require 'ruby-debug'; debugger
```

pry

Installation

Pry is another popular gem for introspecting Ruby code. It is compatible with JRuby. Install `pry` when running a packaged version of `puppetserver` using:

```
$ sudo puppetserver gem install pry --no-ri --no-rdoc
```

Or, if you're running `puppetserver` from source:

```
$ lein gem -c ~/.puppetserver/puppetserver.conf -- install pry \
  --no-ri --no-rdoc
```

Usage

`puppetserver` should be run in the foreground to make use of the `pry` repl. This involves stopping the background service and starting the server in the foreground with the `puppet foreground` subcommand:

```
$ sudo service puppetserver stop
$ sudo puppetserver foreground
```

After installing, you can add a line like this to the Ruby code:

```
require 'pry'; binding.pry
```

This will give you an advanced interactive REPL at the line of code where you've called `pry`.

There are many other gems that are useful for debugging, and a large percentage of them are compatible with JRuby. If you have a favorite that is not mentioned here please let us know, and we will consider adding it to this documentation!

Limitations

We are aware that some favorite gems/tools/features for ruby debugging don't currently work with JRuby/Puppet Server. (For example, some things like color syntax highlighting in Pry.) It's important to us to make sure that the Ruby developer experience is not degraded for developers working via Puppet Server rather than webrick, so, if you run into issues like this, please file an issue on our [Bug Tracker](#), and we will see if it's possible to add support for things that we're missing. In many cases it might be a matter of simply submitting a patch to JRuby, or submitting a JRuby-compatibility patch for an existing gem, and we're interested in trying to help with those sorts of things whenever possible.

Tracing Code Events

Puppet Server can utilize JRuby's standard facilities for tracing events during code execution. For more information on these techniques, see the [Tracing code events](#) on page 171 page.

Running from source

So you'd like to run Puppet Server from source?

The following steps will help you get Puppet Server up and running from source.

Step 0: Quick Start for Developers

```
# clone git repository and initialize submodules
$ git clone --recursive git://github.com/puppetlabs/puppetserver
$ cd puppetserver

# Remove any old config if you want to make sure you're using the latest
# defaults
$ rm -rf ~/.puppetserver

# Run the `dev-setup` script to initialize all required configuration
$ ./dev-setup

# Launch the clojure REPL
$ lein repl
# Run Puppet Server
dev-tools=> (go)
dev-tools=> (help)
```

You should now have a running server. All relevant paths (`$confdir`, `$codedir`, etc.) are configured by default to point to directories underneath `~/.puppetlabs`. These should all align with the default values that puppet uses (for non-root users).

You can find the specific paths in the `dev/puppetserver.conf` file.

In another shell, you can run the agent:

```
# Go to the directory where you checked out puppetserver
$ cd puppetserver
# Set ruby and bin paths
$ export RUBYLIB=./ruby/puppet/lib:./ruby/facter/lib
$ export PATH=./ruby/puppet/bin:./ruby/facter/bin:$PATH
# Run the agent
$ puppet agent -t
```

More detailed instructions follow.

Step 1: Install Prerequisites

Use your system's package tools to ensure that the following prerequisites are installed:

- JDK 1.8 or Java 11
- [Leiningen 2.9.1 \(latest\)](#)
- Git (for checking out the source code)

Step 2: Clone Git Repo and Set Up Working Tree

```
$ git clone --recursive git://github.com/puppetlabs/puppetserver
$ cd puppetserver
```

Step 3: Set up Config Files

The easiest way to do this is to just run:

```
$ ./dev-setup
```

This will set up all of the necessary configuration files and directories inside of your `~/puppetlabs` directory. If you are interested in seeing what all of the default file paths are, you can find them in `./dev/puppetserver.conf`.

The default paths should all align with the default values that are used by puppet (for non-root users).

If you'd like to customize your environment, here are a few things you can do:

- Before running `./dev-setup`, set an environment variable called `MASTERHOST`. If this variable is found during `dev-setup`, it will configure your `puppet.conf` file to use this value for your certname (both for Puppet Server and for puppet) and for the `server` configuration (so that your agent runs will automatically use this hostname as their Puppet primary server).
- Create a file called `dev/user.clj`. This file will be automatically loaded when you run Puppet Server from the REPL. In it, you can define a function called `get-config`, and use it to override the default values of various settings from `dev/puppetserver.conf`. For an example of what this file should look like, see `./dev/user.clj.sample`.

You don't need to create a `user.clj` in most cases; the settings that I change the most frequently that would warrant the creation of this file, though, are:

- `jruby-puppet.max-active-instances`: the number of JRuby instances to put into the pool. This can usually be set to 1 for dev purposes, unless you're working on something that involves concurrency.
- `jruby-puppet.splay-instance-flush`: Do not attempt to splay JRuby flushing, set when testing if using multiple JRuby instances and you need to control when they are flushed from the pool
- `jruby-puppet.master-conf-dir`: the puppet primary server confdir (where `puppet.conf`, `modules`, `manifests`, etc. should be located).
- `jruby-puppet.master-code-dir`: the puppet primary server codedir
- `jruby-puppet.master-var-dir`: the puppet primary server vardir
- `jruby-puppet.master-run-dir`: the puppet primary server rundir
- `jruby-puppet.master-log-dir`: the puppet primary server logdir

Step 4a: Run the server from the clojure REPL

The preferred way of running the server for development purposes is to run it from inside the clojure REPL. The git repo includes some files in the `/dev` directory that are intended to make this process easier.

When running a clojure REPL via the `lein repl` command-line command, lein will load the `dev/dev-tools.clj` namespace by default.

Running the server inside of the clojure REPL allows you to make changes to the source code and reload the server without having to restart the entire JVM. It can be much faster than running from the command line, when you are doing iterative development. We are also starting to build up a library of utility functions that can be used to inspect and modify the state of the running server; see `dev/dev-tools.clj` for more info.

(NOTE: many of the developers of this project are using a more full-featured IDE called [Cursive Clojure](#), built on the IntelliJ IDEA platform, for our daily development. It contains an integrated REPL that can be used in place of the `lein repl` command-line command, and works great with all of the functions described in this document.)

To start the server from the REPL, run the following:

```
$ lein repl
nREPL server started on port 47631 on host 127.0.0.1
dev-tools=> (go)
dev-tools=> (help)
```

Then, if you make changes to the source code, all you need to do in order to restart the server with the latest changes is:

```
dev-tools=> (reset)
```

Restarting the server this way should be significantly faster than restarting the entire JVM process.

You can also run the utility functions to inspect the state of the server, e.g.:

```
dev-tools=> (print-puppet-environment-states)
```

Have a look at `dev-tools.clj` if you're interested in seeing what other utility functions are available.

Step 4b: Run the server from the command line

If you prefer not to run the server interactively in the REPL, you can launch it as a normal process. To start the Puppet Server when running from source, simply run the following:

```
$ lein run -c /path/to/puppetserver.conf
```

Step 4c: Development environment gotchas

Missing git submodules

If you get an error like the following:

```
Execution error (LoadError) at org.jruby.RubyKernel/require
(org/jruby/RubyKernel.java:970).
(LoadError) no such file to load -- puppet
```

Then you've probably forgotten to fetch the git submodules.

Failing tests

If you change the `:webserver :ssl-port` config option from the default value of 8140, tests will fail with errors like the following:

```
lein test :only puppetlabs.general-puppet.general-puppet-int-test/test-external-command-execution

ERROR in (test-external-command-execution) (SocketChannelImpl.java:-2)
Uncaught exception, not in assertion.
expected: nil
2019-02-06 14:58:50,541 WARN [async-dispatch-18] [o.e.j.s.h.ContextHandler]
Empty contextPath
actual: java.net.ConnectException: Connection refused
at sun.nio.ch.SocketChannelImpl.checkConnect (SocketChannelImpl.java:-2)
sun.nio.ch.SocketChannelImpl.finishConnect (SocketChannelImpl.java:717)
org.apache.http.impl.nio.reactor.DefaultConnectingIOReactor.processEvent
(DefaultConnectingIOReactor.java:171)

org.apache.http.impl.nio.reactor.DefaultConnectingIOReactor.processEvents
(DefaultConnectingIOReactor.java:145)
```



```
org.apache.http.impl.nio.reactor.AbstractMultiworkerIOReactor.execute
(AbstractMultiworkerIOReactor.java:348)

org.apache.http.impl.nio.conn.PoolingNHttpClientConnectionManager.execute
(PoolingNHttpClientConnectionManager.java:192)
org.apache.http.impl.nio.client.CloseableHttpAsyncClientBase$1.run
(CloseableHttpAsyncClientBase.java:64)
java.lang.Thread.run (Thread.java:844)
```

Changing the `ssl-port` variable back to 8140 makes the tests run properly.

Running the Agent

Use a command like the one below to run an agent against your running puppetserver:

```
puppet agent --confdir ~/.puppetlabs/etc/puppet \
--debug -t
```

Note that a system installed Puppet Agent is ok for use with source-based PuppetDB and Puppet Server. The `--confdir` above specifies the same confdir that Puppet Server is using. Because the Puppet Agent and Puppet Server instances are both using the same confdir, they're both using the same certificates as well. This alleviates the need to sign certificates as a separate step.

Running the Agent inside a Docker container

You can easily run a Puppet Agent inside a Docker container, either by using the `host` network profile or by accessing the Puppetserver service using the Docker host IP:

```
docker run -ti \
--name agent1 \
puppet/puppet-agent-ubuntu \
agent -t --server 172.17.0.1
```

```
docker run -ti \
--name agent2 \
--network host \
--add-host puppet:127.0.0.1 \
puppet/puppet-agent-ubuntu
```

To start another Puppet Agent run in a previous container you can use the `docker start` command:

```
docker start -a agent1
```

Running tests

- `lein test` to run the clojure test suite
- `rake spec` to run the jruby test suite

The Clojure test suite can consume a lot of transient memory. Using a larger JVM heap size when running tests can significantly improve test run time. The default heap size is somewhat conservative: 1 GB for the minimum heap (much lower than that as a maximum can lead to Java `OutOfMemory` errors during the test run) and 2 GB for the maximum heap. While the heap size can be configured via the `-Xms` and `-Xmx` arguments for the `:jvm-opts` `defproject` key within the `project.clj` file, it can also be customized for an individual user environment via either of the following methods:

1. An environment variable named `PUPPETSERVER_HEAP_SIZE`. For example, to use a heap size of 5 GiB for a `lein test` run, you could run the following:

```
$ PUPPETSERVER_HEAP_SIZE=6G lein test
```

2. A `lein profiles.clj` setting in the `:user` profile under the `:puppetserver-heap-size` key. For example, to use a heap size of 5 GiB, you could add the following key to your `~/ .lein/profiles.clj` file:

```
{:user {:puppetserver-heap-size "5G"
      ...}}
```

With the `:puppetserver-heap-size` key defined in the `profiles.clj` file, any subsequent `lein test` run would utilize the associated value for the key. If both the environment variable and the `profiles.clj` key are defined, the value from the environment variable takes precedence. When either of these settings is defined, the value is used as both the minimum and maximum JVM heap size.

Installing Ruby Gems for Development

The gems that are vendored with the puppetserver OS packages will be automatically installed into your dev environment by the `./dev-setup` script. If you wish to install additional gems, please see the [Using Ruby gems](#) on page 138 document for detailed information.

Debugging

For more information about debugging both Clojure and JRuby code, please see [Developer debugging](#) on page 164 documentation.

Running PuppetDB

To run a source PuppetDB with Puppet Server, Puppet Server needs standard PuppetDB configuration and how to find the PuppetDB terminus. First copy the `dev/puppetserver.conf` file to another directory. In your copy of the config, append a new entry to the `ruby-load-path` list: `<PDB source path>/puppet/lib`. This tells PuppetServer to load the PuppetDB terminus from the specified directory.

From here, the instructions are similar to installing PuppetDB manually via packages. The PuppetServer instance needs configuration for connecting to PuppetDB. Instructions on this configuration are below, but the official docs for this can be found [here](#).

Update `~/ .puppetlabs/etc/puppet/puppet.conf` to include:

```
[server]
storeconfigs = true
storeconfigs_backend = puppetdb
reports = store,puppetdb
```

Create a new puppetdb config file `~/ .puppetlabs/etc/puppet/puppetdb.conf` that contains

```
[main]
server_urls = https://<SERVERHOST>:8081
```

Then create a new routes file at `~/ .puppetlabs/etc/puppet/routes.yaml` that contains

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

Assuming you have a PuppetDB instance up and running, start your Puppet Server instance with the new `puppetserver.conf` file that you changed:

```
lein run -c ~/<YOUR CONFIG DIR>/puppetserver.conf
```

Depending on your PuppetDB configuration, you might need to change some SSL config. PuppetDB requires that the same CA that signs its certificate, also has signed Puppet Server's certificate. The easiest way to do this is to point

PuppetDB at the same configuration directory that Puppet Server and Puppet Agent are pointing to. Typically this setting is specified in the `jetty.ini` file in the PuppetDB `conf.d` directory. The update would look like:

```
[jetty]

#...
ssl-cert = <home dir>/etc/puppetlabs/etc/puppet/ssl/certs/<SERVERHOST>.pem
ssl-key = <home dir>/etc/puppetlabs/etc/puppet/ssl/private_keys/
<SERVERHOST>.pem
ssl-ca-cert = <home dir>/etc/puppetlabs/etc/puppet/ssl/certs/ca.pem
```

After the SSL config is in place, start (or restart) PuppetDB:

```
lein run services -c <path to PDB config>/conf.d
```

Then run the Puppet Agent and you should see activity in PuppetDB and Puppet Server.

Tracing code events

The JRuby runtime supports the Ruby `set_trace_func` Kernel method for tracing code events, e.g., lines of code being executed and calls to C-language routines or Ruby methods. This can likewise be used in Puppet Server for tracing.

In order to enable a more verbose level of tracing, e.g., to capture lower-level calls into C code, the `jruby.debug.fullTrace` Java property must be set to "true". If you are running Puppet Server from source, this can be done by adding the option to the `project.clj` file:

```
:jvm-opts [ "-Djruby.debug.fullTrace=true" ]
```

If you are running Puppet Server from a package, this can be done by adding the option to the `puppetserver` file in `/etc/sysconfig` or `/etc/default`, depending upon your OS distribution:

```
JAVA_ARGS="-Xms2g -Xmx2g -Djruby.debug.fullTrace=true"
```

A call to the `set_trace_func` function can be done in one of the Ruby files in the Puppet Server code. For the trace to be in effect for the full execution of Ruby code, one common place to put this call would be at the top of the `../src/ruby/puppetserver-lib/puppet/server/master.rb` file, which configures Ruby Puppet for running inside Puppet Server. A basic implementation might look like this:

```
set_trace_func proc { |event, file, line, id, binding, classname|
  printf "%8s %s:%-2d %10s %8s\n", event, file, line, id, classname
}
```

Note that `printf` will write each trace line to stdout. If you are running Puppet Server from a package install, stdout should be routed to the `/var/log/puppetserver-daemon.log` file.

Lines of output from `set_trace_func` look like the following:

```
c-call /usr/share/puppetserver/puppet-server-release.jar!/META-INF/
jruby.home/lib/ruby/shared/jopenssl19/openssl/ssl-internal.rb:30 initialize
OpenSSL::X509::Store
```

You could use this technique to locate any references made to specific class names from code and the active stack at the point of each reference. For example, to locate callers of any OpenSSL classes, you could add the following to the `set_trace_func` call:

```
set_trace_func proc { |event, file, line, id, binding, classname|
  if classname.to_s =~ /OpenSSL/
    printf "%8s %s:%-2d %10s %8s\n", event, file, line, id, classname
    puts caller
  end
end
```

}

Puppet Server HTTP API

- [Puppet Server HTTP API overview](#) on page 172
 - [PSON](#)
 - [Schemas \(JSON\)](#) on page 185
- These JSON files contain schemas for the various HTTP API objects

Puppet Server HTTP API overview

Puppet Server provides several services via HTTP API, and the Puppet agent application uses those services to resolve a node's credentials, retrieve a configuration catalog, retrieve file data, and submit reports.

Puppet and Puppet CA APIs

Puppet Server's HTTP API is split into two separately versioned APIs:

- An API for configuration-related services
- An API for the certificate authority (CA).

All configuration endpoints are prefixed with `/puppet`, while all CA endpoints are prefixed with `/puppet-ca`. All endpoints are explicitly versioned: the prefix is always immediately followed by a string like `/v3` (a directory separator, the letter `v`, and the version number of the API).

Authorization

Authorization for `/puppet` and `/puppet-ca` endpoints is controlled with [auth.conf](#) on page 78.

Puppet V3 HTTP API

The Puppet agent application uses several network services to manage systems. These services are all grouped under the `/puppet` API. Other tools can access these services and use the Puppet primary server's data for other purposes.

The V3 API contains endpoints of two types: those that are based on dispatching to Puppet's internal "indirector" framework, and those that are not (namely the [environments endpoint](#)).

Every HTTP endpoint that dispatches to the indirector follows the form `/puppet/v3/:indirection/:key?environment=:environment`, where:

- `:environment` is the name of the environment that should be in effect for the request. Not all endpoints need an environment, but the query parameter must always be specified.
- `:indirection` is the indirection to which the request is dispatched.
- `:key` is the "key" portion of the indirection call.

Using this API requires significant understanding of how Puppet's internal services are structured, but the following documents specify what is available and how to interact with it.

Configuration management services

The Puppet agent application directly uses these services to manage the configuration of a node.

These endpoints accept payload formats formatted as JSON by default (MIME type of `application/json`), except for `File Content` and `File Bucket File`, which always use `application/octet-stream`.

Note: Legacy PSON (MIME type of `text/pson`) is still an available format, but should be used only as a fallback for binary content.

- [Facts](#)
- [Catalog](#)
- [Node](#)
- [File bucket file](#)
- [File content](#)

- [File metadata](#)
- [Report](#)

Environments endpoint

The `/puppet/v3/environments` endpoint uses a different format than the configuration management and informational services endpoints.

The endpoint accepts only payloads formatted as JSON, and responds with JSON (MIME type of `application/json`).

- [Environments](#)

Puppet Server-specific endpoints

Puppet Server adds several unique endpoints of its own. They include these additional `/puppet/v3/` endpoints:

- [Environment classes](#) on page 192, at `/puppet/v3/environment_classes`
- [Environment modules](#) on page 198, at `/puppet/v3/environment_modules`
- [Static file content](#) on page 200, at `/puppet/v3/static_file_content`

It also includes these unique APIs, with endpoints containing other URL prefixes:

- [Services endpoint](#) on page 189, at `/status/v1/services`
- [v1 metrics](#) on page 185, at `/metrics/v1/mbeans`
- [v2 \(Jolokia\) metrics](#) on page 186, at `/metrics/v2/`
- Admin API, at `/puppet-admin-api/v1/`:
 - [Environment cache](#) on page 202, at `/puppet-admin-api/v1/environment-cache`
 - [JRuby pool](#) on page 202, at `/puppet-admin-api/v1/jruby-pool`

Error responses

The `environments` endpoint responds to error conditions in a uniform manner and uses standard HTTP response codes to signify those errors.

Request problem	HTTP API error response code
Client submits malformed request	400 Bad Request
Unauthorized client	403 Not Authorized
Client uses an HTTP method not permitted for the endpoint	405 Method Not Allowed
Client requests a response in a format other than JSON	406 Unacceptable
Server encounters an unexpected error while handling a request	500 Server Error
Server can't find an endpoint handler for an HTTP request	404 Not Found

Except for HEAD requests, error responses contain a body of a uniform JSON object with the following properties:

- `message`: (String) A human-readable message explaining the error.
- `issue_kind`: (String) A unique label to identify the error class.

Puppet provides a [JSON schema for error objects](#). Endpoints implemented by Puppet Server have a different error schema:

```
{
  "msg": "",
  "kind": ""
}
```

CA V1 HTTP API

The certificate authority (CA) API contains all of the endpoints supporting Puppet's public key infrastructure (PKI) system.

The CA V1 endpoints share the same basic format as the Puppet V3 API, because they are based on the interface of Puppet's indirector-based CA. However, Puppet Server's CA is implemented in Clojure. Both have a different prefix and version than the V3 API.

These endpoints follow the form `/puppet-ca/v1/:indirection/:key`, where:

- `:indirection` is the indirection to which the request is dispatched.
- `:key` is the "key" portion of the indirection call.

As with the Puppet V3 API, using this API requires a significant amount of understanding of how Puppet's internal services are structured. The following documents specify what is available and how to interact with it.

SSL certificate-related services

These endpoints accept only plain-text payload formats. Historically, Puppet has used the MIME type `s` to mean `text/plain`. In Puppet 5, it always uses `text/plain`, but continues to accept `s` as an equivalent.

- [Certificate](#) on page 175
- [Certificate Request](#) on page 177
- [Certificate Status](#) on page 179
- [Certificate Revocation List](#) on page 182

Serialization formats

Puppet sends messages using several serialization formats. Not all REST services support all of the formats.

- [JSON](#)
- [PSON](#)

Puppet v3 API

Puppet v4 API

Catalog API

The catalog API returns a compiled catalog for the node specified in the request, making use of provided metadata like facts or environment if specified. If not specified, it will attempt to fetch this data from Puppet's configured sources (usually PuppetDB or a node classifier). The returned catalog is in JSON format, ready to be parsed and applied by an agent.

POST /puppet/v4/catalog

(Introduced in Puppet Server 6.3.0)

The input data for the catalog to be compiled is submitted as a JSON body with the following form:

```
{
  "certname": "<node name>",
  "persistence": { "facts": <true/false>, "catalog": <true/false> },
  "environment": "<environment name>",
  # The rest are optional:
  "facts": { "values": { "<fact name>": <fact value>, ... } },
  "trusted_facts": { "values": { "<fact name>": <fact value>, ... } },
  "transaction_uuid": "<uuid string>",
  "job_id": "<id string>",
  "options": { "prefer_requested_environment": <true/false>,
               "capture_logs": <true/false>,
               "log_level": <err/warning/info/debug> }
}
```

`certname` (required)

The name of the node for which to compile the catalog.

`persistence` (required)

A hash containing two required keys, `facts` and `catalog`, which when set to `true` will cause the facts and reports to be stored in PuppetDB, or discarded if set to `false`.

`environment` (required)

The name of the environment for which to compile the catalog. If `prefer_requested_environment` is `true`, override the classified environment with this param. If it is `false`, only respect this if the classifier allows an agent-specified environment.

`facts`

A hash with a required `values` key, containing a hash of all the facts for the node. If not provided, Puppet will attempt to fetch facts for the node from PuppetDB.

`trusted_facts`

A hash with a required `values` key containing a hash of the trusted facts for a node. In a normal agent's catalog request, these would be extracted from the cert, but this endpoint does not require a cert for the node whose catalog is being compiled. If not provided, Puppet will attempt to fetch the trusted facts for the node from PuppetDB or from the provided facts hash.

`transaction_uuid`

The id for tracking the catalog compilation and report submission.

`job_id`

The id of the orchestrator job that triggered this run.

`options`

A hash of options beyond direct input to catalogs.

`prefer_requested_environment` Whether to always override a node's classified environment with the one supplied in the request. If this is `true` and no environment is supplied, fall back to the classified environment, or finally, `'production'`.

`capture_logs` Whether to return the errors and warnings that occurred during compilation alongside the catalog in the response body.

`log_level` The logging level to use during the compile when `capture_logs` is `true`. Options are `'err'`, `'warning'`, `'info'`, and `'debug'`.

Schema

The catalog response body conforms to the [catalog schema](#).

Authorization

All requests made to the catalog API are authorized using the Trapperkeeper-based `auth.conf`. For more information about the Puppet Server authorization process and configuration settings, see the [auth.conf](#) on page 78.

CA v1 API Certificate

The `certificate` endpoint returns the certificate for the specified name, which might be either a standard certname or `ca`.

Find

Get a certificate.

```
GET /puppet-ca/v1/certificate/:nodename
```

Supported HTTP Methods

GET

Supported Response Formats

text/plain

The returned certificate is always in the .pem format.

Parameters

None

Responses

Certificate found

```
GET /puppet-ca/v1/certificate/elmo.mydomain.com

HTTP 200 OK
Content-Type: text/plain

-----BEGIN CERTIFICATE-----
MIIFujCCA6KgAwIBAgIBATANBgkqhkiG9w0BAQsFADBiMWAwXgYDVQQDDFdQdXBw
ZXQgQ0EgZ2VuZuZXJhdGVkIG9uIGRoY3A1MC5reWxvLmJhY2tsaW5lLnB1cHBldGxh
YnMubmV0IGF0IDIwMTMtMDYtMjQgMTY6MzA6MTcgLTA3MDAwHhcNMjMwNjIzMDYt
MDE5WWhcNMjMwNjIzMDYtMDE5WjBiMWAwXgYDVQQDDFdQdXBwZXQgQ0EgZ2VuZuZXJh
dGVkIG9uIGRoY3A1MC5reWxvLmJhY2tsaW5lLnB1cHBldGxhYnMubmV0IGF0IDIw
MTMtMDYtMjQgMTY6MzA6MTcgLTA3MDAwggIiMA0GCSqGSIb3DQEBAQUAA4ICDwAw
ggIKAoICAQDABqllmzccjuRmnCdXvTmdexJGlb9S8r8+I+G6fkHTa1WKDSob9PZpS
eXJtanbl0zNws9yBt1Dko2zhKDKctBRWf5CT42nDxBZPY7SaD7KaCzb07g9wfWgU
BOB/6smyl/iySEmQzzFLRgZbo5A9WLiY/UdyQimlfaakevRme2Xi/l/i0TKbpu27
DhCS+E8aC8Bvaj0ph0T+TzYphTR76pP5Kps6G7Jyk/HFYrVXnY44X2Pet2mgkEXp
xHCbU+qCFMtTLMG+ZArA/noM3I/O6W5LhLSzApjut/M7UdMlpZ45PGDrsvf2R306
NcOh+zbkbkxuaIaGqaxeaeYzb01A3gXhZvYaV6EKjXNtm7BslpsvhLi0U+CWyb3C
qRkpx0MgxJgxoqViJ4TDVA+EmztOnK86+G4HGeJqTPQloYO/TdlwMT1Txh9T5Ue
Wctw/g+4o22EyJQRo+vxxzHNRife7EHAerMUtLT5u9MJeQb9NliUR2ATNAN+QiB2
KEqyc9eMapK6QUZfV23Xvbdup1WCrgsWXBqyRWKV7x0sc9Wv8RMRKEFYaBeHEVXU
m0hGgF34Z8Rzphq2H1FjkLD+xbtG0jrAlMb2De81Hfvrf18497X5UMPtsuzOt/XU
PHbbSCy+05J7VNZ/gaiGqgpHfcG5yiqCdjlLIzhFuuvvm+fADPxK38wIDAQABO3sw
eTA3BglghkgBhvhCAQ0EKhyOUHVwcGV0IFJlYnkvT3BlblNTTCBjbnRlcm5hbCBD
ZXJ0aWZlY2F0ZTA0BGNVHQ8BAf8EBAMCAQYwDwYDVR0TAQH/BAUwAwEB/zAdBgNV
HQ4EFggQUEhn/MqSDtuxgl2klWosCGenxf1cwDQYJKoZIhvcNAQELBQADggIBAHLG
L3FG/keKlGqs70PxxvRlwCo4VM3K/C+5uxnzmlMHEAd96nhtwE6YskUe+XgDiXfC
+NXS2C4TeTQAEo6grREapWDjhJvrhrgqTzmb4lTKzb91II3/VGYzG5UXxID262zy
QLoX/IBN/xDJ5ds0wF2adUbnHUSsEGGLjgngewH/7kjeW/L5iL+USXZnKHPSggjM
RAEjluCE/rDqDN0xhOS4K2PjseFm7krW4cZ0gNmxdRhC70hmJ56dH92F4M9jn7Qy
EqxWB304U/aMc03NJxTQc7Arel/pUtjtI6hxM4miHbjSh6RfNBqhzRyJvxA6gc6g
m3kumdw04KZFSS/6fPFFbI60i5K+vioB4CnUWpj+3Z+OnDEVhQJEACR1JC8A67Ih
x+GDlbHLU1BwonwZzSMJz+ABXV3dwIrOSFHI0UmDXg+cIdZ+SaL93qMjUVU4v9nu
gR9yJGMqNuzLjgfbD/KGCEEAITKBwPvCVd//OMlWVrXr7vvt+yo6STIltJxABJDp
CSLyHUtT++CsPXsPADxgRctpIbhlMFEivkK9Oy+W/CZYIZnARVysUpMWg7TkXqx
mSCXy9ZXLWqU/ssVhbLS9vFVa5pvxcyfiRpsFg0XZsx8mnZP6OaWcL8FjF+/NwNP
tg1+DuYTn+d540Hi/GZEnvutgrDZyrJDrrb/Czm9
-----END CERTIFICATE-----
```

Certificate not found

```
GET /puppet-ca/v1/certificate/certificate_does_not_exist

HTTP 404 Not Found
Content-Type: text/plain

Not Found: Could not find certificate certificate_does_not_exist
```


No Certificate name given

```
GET /puppet-ca/v1/certificate

HTTP/1.1 400 Bad Request
Content-Type: text/plain

No request key specified in /puppet-ca/v1/certificate
```

Primary Server is not a CA

```
GET /puppet-ca/v1/certificate/valid_certificate

HTTP/1.1 400 Bad Request
Content-Type: text/plain

this server is not a CA
```

Certificate Request

The `certificate_request` endpoint submits a Certificate Signing Request (CSR) to the primary server. CSRs that have been submitted can then also be retrieved. The returned CSR is always in the `.pem` format.

Find

Get a submitted CSR

```
GET /puppet-ca/v1/certificate_request/:nodename
Accept: text/plain
```

Save

Submit a CSR

```
PUT /puppet-ca/v1/certificate_request/:nodename
Content-Type: text/plain
```

Note: The `:nodename` must match the Common Name on the submitted CSR.

Note: Although the `Content-Type` is sent as `text/plain` the content is specifically a CSR in PEM format.

Search

Note: The plural `certificate_requests` endpoint is a legacy feature. Puppet Server doesn't support it, and we don't plan to add support in the future.

List submitted CSRs

```
GET /puppet-ca/v1/certificate_requests/:ignored_pattern
Accept: text/plain
```

The `:ignored_pattern` parameter is not used, but must still be provided.

Destroy

Delete a submitted CSR

```
DELETE /puppet-ca/v1/certificate_request/:nodename
Accept: text/plain
```

Supported HTTP Methods

The default configuration only allows requests that result in a Find and a Save. You need to modify `auth.conf` in order to allow clients to use Search and Destroy actions. It is not recommended that you change the default settings.

GET, PUT, DELETE

Supported Response Formats

text/plain

The returned CSR is always in the .pem format.

Parameters

None

Examples

CSR found

```
GET /puppet-ca/v1/certificate_request/agency

HTTP/1.1 200 OK
Content-Type: text/plain

-----BEGIN CERTIFICATE REQUEST-----
MIIBnzCCAQwCAQAwYzELMAkGALUEBhMCVUsxDzANBgNVBAgTBkxvbmRvbJEPMA0G
A1UEBxMGTG9uZG9uMSEwHwYDVQQKEzhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQx
DzANBgNVBAMTBmFnZW5jeTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAxSCr
FKUKjVGFPUQ0iGM9mZKw94sOIgGohqrHH743kPvjsId3d38Qk+H+1DbVf42bQY0W
kAVcwNDqmBnx0lOtQ0oeGnbbwlJFjhqXr8jFELjPrc9S2/IIILDf/FeYWw9lRiOV
LoU6ZfCIBfq6v4D4KX3utRbOoELNyBeT6VA1ufMCAwEAAaAAMakGBSsOAuIPBQAD
gYEAno7O1jkr56TNMe1Cw/eyQUIaniG22+0kmofTjlcMYZ/IKCOz+HRgnDtBPf8j
O5nt0PQN8YClW7Xx2U8ZTvBXn/UEKmtCBkbF+SULiayxPgfyKy/axinfutEChnHS
ZtUMUBLlh+gGFqOuH69979SJ2QmQC6FNomTkYI7FOHD/TG0=
-----END CERTIFICATE REQUEST-----
```

CSR not found

```
GET /puppet-ca/v1/certificate_request/does_not_exist

HTTP/1.1 404 Not Found
Content-Type: text/plain

Not Found: Could not find certificate_request does_not_exist
```

No node name given

```
GET /puppet-ca/v1/certificate_request

HTTP/1.1 400 Bad Request
Content-Type: text/plain

No request key specified in /puppet-ca/v1/certificate_request
```

Delete a CSR that exists

```
DELETE /puppet-ca/v1/certificate_request/agency
Accept: text/plain

HTTP/1.1 200 OK
Content-Type: text/plain

1
```

Delete a CSR that does not exist

```
DELETE /puppet-ca/v1/certificate_request/missing
```

```
Accept: text/plain

HTTP/1.1 200 OK
Content-Type: text/plain

false
```

Retrieve all CSRs

```
GET /puppet-ca/v1/certificate_requests/ignored
Accept: text/plain

HTTP/1.1 200 OK
Content-Type: text/plain

-----BEGIN CERTIFICATE REQUEST-----
MIIBNzCCAQAwYzELMAkGA1UEBhMCVUsxDzANBgNVBAGTBKxvbmRvbJEPMA0G
A1UEBxMGTG9uZG9uMSEwHwYDVQQKEWhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQx
DzANBgNVBAMTBmFnZW5jeTTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAxScR
FKUKjVGFPUQ0iGM9mZKw94sOIgGohqrHH743kPvjsId3d38Qk+H+1DbVf42bQY0W
kAVcwNDqmBnx010tQ0oeGnbbwlJFjhqXr8jFEljPrc9S2/IIILdf/FeYWw9lRiOV
LoU6ZfCIBfq6v4D4KX3utRbOoELNyBeT6VA1ufMCAwEAAaAAMakGBSsOAwIPBQAD
gYEAno701jKR56TNMe1Cw/eyQUIaniG22+0kmoftjlcMYZ/IKCOz+HRgnDtBPf8j
05nt0PQN8YClW7Xx2U8ZTvbXn/UEKMTcBkbF+SULiayxPgfyKy/axinfutEChnHS
ZtUMUBLlh+gGFqOuH69979SJ2QmQC6FNomTkYI7FOHD/TG0=
-----END CERTIFICATE REQUEST-----

---
-----BEGIN CERTIFICATE REQUEST-----
MIIBNjCCAQsCAQAwYjELMAkGA1UEBhMCVUsxDzANBgNVBAGTBKxvbmRvbJEPMA0G
A1UEBxMGTG9uZG9uMSEwHwYDVQQKEWhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQx
DjAMBgNVBAMTBWFnZW50MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCltucK
enT1CkDPgsCU/0e2cbzRsiKF8yHH7+ntF6Q3d9ZCaZWJ00mj0+YmiYrnum+KAiKE
45Iaf9vaUV3CPsDVRUPOI8kYehiv868ZhP3nxblE6iuNBK+Fdv9GN/vKQrmL5iRE
bIrOM3/lxps7SpidGdA6EIVlS3604bwLY4xHNQIDAQABAAwCQYFKw4DAg8FAAOB
gQAXH0YFuidPqB6P2MyPEEGZ3rzoZINBx/oXvGptXI60Zy5mgH6iAkrZfi57pEzP
jFoO2JRaFXTJC1FVpc4zRlK6sq4h3fIMwqppJRX+5wJNKyhU61eY2gR20/rAJzw4
wcUKf9JhoE7/plcUulIIIq7t/ibCvf0LYSFwGqTwGqN2TQ==
-----END CERTIFICATE REQUEST-----
```

The CSR PEMs are separated by "\n---\n"

Certificate Status

The `certificate status` endpoint allows a client to read or alter the status of a certificate or pending certificate request.

Note: */puppet-ca/v1 API requires certificate-based authentication. The certificate used must have the `pp_cli_auth` extension. This extension is on the server's certificate by default so `curl` (and `puppetserver ca CLI`) calls from the same host as the CA are designed to work out of the box. Using `curl` from a different host requires issuing a special certificate.*

The following is an example of how to interact with this API:

```
curl -G --cert $(puppet config print hostcert) --key $(puppet config
print hostprivkey) --cacert $(puppet config print cacert) https://<PE
Server>:8140/puppet-ca/v1/certificate_status/<Node>
```

Find

```
GET /puppet-ca/v1/certificate_status/:certname
Accept: application/json, text/pson
```

Retrieve information about the specified certificate. Similar to `puppetserver ca list --certname <certname>`.

Search

```
GET /puppet-ca/v1/certificate_statuses/:any_key?state=:state
Accept: application/json, text/pson
```

Retrieve information about all known certificates. Similar to `puppetserver ca list --all`. A key is required but is ignored.

Parameters

- `state` (optional): The certificate state by which to filter search results. Valid states are 'requested', 'signed', and 'revoked'.

Save

```
PUT /puppet-ca/v1/certificate_status/:certname
Content-Type: text/pson
```

Change the status of the specified certificate. The desired state is sent in the body of the PUT request as a one-item PSON hash; the two allowed complete hashes are:

- `{ "desired_state": "signed" }` (for signing a certificate signing request, similar to `puppetserver ca sign`). To set the validity period of the signed certificate, specify the `cert_ttl` key in the body of the request, with an integer value. By default, this key specifies the number of seconds, but you can specify another time unit. See [configuration](#) for a list of Puppet's accepted time unit markers.
- `{ "desired_state": "revoked" }` (for revoking a certificate, similar to `puppetserver ca revoke`).

Note that revoking a certificate does not clean up other info about the host; see the DELETE request for more information.

Delete

```
DELETE /puppet-ca/v1/certificate_status/:hostname
Accept: application/json, text/pson
```

Cause the certificate authority to discard all SSL information regarding a host (including any certificates, certificate requests, and keys). This does not revoke the certificate if one is present; if you wish to emulate the behavior of `puppet cert --clean`, you must PUT a `desired_state` of `revoked` before deleting the host's SSL information.

Note: the [Certificate Clean](#) on page 192 API can be used to accomplish both revoking and cleaning in one request.

If the deletion was successful, it returns a string listing the deleted classes like

```
"Deleted for myhost: Puppet::SSL::Certificate, Puppet::SSL::Key"
```

Otherwise it returns

```
"Nothing was deleted"
```

Supported HTTP Methods

This endpoint is disabled in the default configuration. It is recommended to be careful with this endpoint, as it can allow control over the certificates used by the puppet primary server.

GET, PUT, DELETE

Supported Response Formats

```
application/json, text/plain, */*
```

Examples

Certificate information

[illegible]

Search unsigned certs (CSRs)

```
GET /puppet-ca/v1/certificate_statuses/ignored?state=requested
HTTP/1.1 200 OK
Content-Type: text/pson

[
  {
    "name": "mycertname1",
    "state": "requested",

    "fingerprint": "A6:44:08:A6:38:62:88:5B:32:97:20:49:8A:4A:4A:AD:65:C3:3E:A2:4C:30:72:73:02:00",
    "fingerprints": {

      "default": "A6:44:08:A6:38:62:88:5B:32:97:20:49:8A:4A:4A:AD:65:C3:3E:A2:4C:30:72:73:02:00",
      "SHA1": "77:E6:5A:7E:DD:83:78:DC:F8:51:E3:8B:12:71:F4:57:F1:C2:34:AE",
      "SHA256": "A6:44:08:A6:38:62:88:5B:32:97:20:49:8A:4A:4A:AD:65:C3:3E:A2:4C:30:72:73:02:00:CA:A0:8C:B9:FE:9D:C2:72:18:57:08:E9:4B:11:B7:BC:4E:F7:52:C8:9C:76:03:45:B4:B0",
      "SHA512": "CA:A0:8C:B9:FE:9D:C2:72:18:57:08:E9:4B:11:B7:BC:4E:F7:52:C8:9C:76:03:45:B4:B0"
    },
    "dns_alt_names": [ ]
  },
  {
    "name": "mycertname2",
    "state": "requested",

    "fingerprint": "A6:44:08:A6:38:62:88:5B:32:97:20:49:8A:4A:4A:AD:65:C3:3E:A2:4C:30:72:73:02:00",
    "fingerprints": {

      "default": "A6:44:08:A6:38:62:88:5B:32:97:20:49:8A:4A:4A:AD:65:C3:3E:A2:4C:30:72:73:02:00",
      "SHA1": "77:E6:5A:7E:DD:83:78:DC:F8:51:E3:8B:12:71:F4:57:F1:C2:34:AE",
      "SHA256": "A6:44:08:A6:38:62:88:5B:32:97:20:49:8A:4A:4A:AD:65:C3:3E:A2:4C:30:72:73:02:00:CA:A0:8C:B9:FE:9D:C2:72:18:57:08:E9:4B:11:B7:BC:4E:F7:52:C8:9C:76:03:45:B4:B0",
      "SHA512": "CA:A0:8C:B9:FE:9D:C2:72:18:57:08:E9:4B:11:B7:BC:4E:F7:52:C8:9C:76:03:45:B4:B0"
    }
  }
]
```

```

"SHA256" : "A6:44:08:A6:38:62:88:5B:32:97:20:49:8A:4A:4A:AD:65:C3:3E:A2:4C:30:72:73:02:C5"
"SHA512" : "CA:A0:8C:B9:FE:9D:C2:72:18:57:08:E9:4B:11:B7:BC:4E:F7:52:C8:9C:76:03:45:B4:B0"
    },
    "dns_alt_names" : [ ]
  }
]

```

Revoking a certificate

```

PUT /puppet-ca/v1/certificate_status/mycertname HTTP/1.1
Content-Type: text/pson
Content-Length: 27

{"desired_state":"revoked"}

```

This has no meaningful return value.

Deleting the certificate information

```
DELETE /puppet-ca/v1/certificate_status/mycertname HTTP/1.1
```

Gets the response:

```
"Deleted for mycertname: Puppet::SSL::Certificate, Puppet::SSL::Key"
```

Certificate Revocation List

The `certificate_revocation_list` endpoint retrieves a Certificate Revocation List (CRL) from the primary server. The returned CRL is always in the `.pem` format.

Find

Get the submitted CRL

```
GET /puppet-ca/v1/certificate_revocation_list/ca
Accept: text/plain
```

Supported HTTP Methods

GET

Supported Response Formats

text/plain

The returned CRL is always in the `.pem` format.

Parameters

None

Examples

Because the returned CRL always looks similar to the human eye, the successful examples are each followed by an openssl decoding of the CRL PEM file.

Empty revocation list

```

GET /puppet-ca/v1/certificate_revocation_list/ca

HTTP/1.1 200 OK
Content-Type: text/plain

```

```

-----BEGIN X509 CRL-----
MIICdzBhAgEBMA0GCSqGSIb3DQEBBQUAMBA8xHTAbBgNVBAMMFBIcHBldCBDQTog
bG9jYXxob3N0Fw0xMzA3MTYyMDQ4NDJhZjAwODAwMDQ4NDNaoA4wDDAKBgNV
HRQEAwIBADANBgkqhkiG9w0BAQUFAAOCAgEAQyBJOy3dtCOcrb0Fu7Z00idQnarg
IzXUV/ugldauPEVYURLNNr+CJrr89QZnU/71lqggPTN/J47mO/lffMSPjmINE+ng
XzOfm0qCG2+gNyaOBODemQTLdHPIIXvcm7T+wEqc7XFW2tjEdpEubZgweruU/+DB
RX6/PhFbalQ0bKcMeFLzLAD4mmtBaQCJISmUUFWxlpyCS6pgBtQ1bNy3PJPN2PNW
YpDf3DNZ16vrAJ4a4SzXLXCoONw0MGxZcS6/hctJ75Vz+dTMrArKwckytWgQS/5e
c/1/wlMzn4xlho+EcIPMPfCB5hW1qzGU2WjUakTVxzF4goamnfFuKbHKEoXVOo9C
3dEQ9un4UydlxHxj8WvQck79In5/S2l9hdqp4eud4BaYB6tNRKxlUntSCvCNriR2
wrDNsMuQ5+KJReG51vM0OzzKmlScgIHagbVeNFZi9X6Tps02bLEZX2xyqKw4xrre
OIEZRoJrmX3VQ/4u9hj14Qbt72/khYo6z/Fckc5zVD+dW4fjP2ztVTSPzBqIK3+H
zAgewYW6cJ6Aan8GS13IfRqj6W1OubWj8Gr1U0dOE7SkBX6w/X6luqsHrOyg/E/Z
0Wcz/V+W5izxa4Spm0x4sfpNzf/bNmjTe4M2MXyn/hXx5MdHf/HZdhOs/lzwKUGL
kEwcy38d6hYtUjs=
-----END X509 CRL-----

```

```

> openssl crl -inform PEM -in empty.crl -text -noout
Certificate Revocation List (CRL):

```

```

    Version 2 (0x1)
    Signature Algorithm: sha1WithRSAEncryption
    Issuer: /CN=Puppet CA: localhost
    Last Update: Jul 16 20:48:42 2013 GMT
    Next Update: Jul 15 20:48:43 2018 GMT
    CRL extensions:
        X509v3 CRL Number:

```

```

    0
No Revoked Certificates.

```

```

    Signature Algorithm: sha1WithRSAEncryption
    ab:20:49:3b:2d:dd:b4:23:9c:ad:bd:05:bb:b6:4e:3a:20:d0:
    9d:aa:e0:23:35:d4:57:fb:a0:d5:d6:ae:3c:45:72:51:12:cd:
    36:bf:82:26:ba:fc:f5:06:67:53:fe:f5:96:a8:29:59:33:7f:
    27:8e:e6:3b:f9:5f:7c:c4:8f:8e:62:0d:13:e9:e0:5f:33:9f:
    7e:6d:2a:08:6d:be:80:dc:9a:38:13:9d:12:64:13:2d:d1:cf:
    21:7b:dc:9b:b4:fe:c0:4a:9c:ed:71:56:da:d8:c4:76:91:2e:
    6d:98:30:7a:bb:94:ff:e0:c1:45:7e:bf:3e:11:5b:6a:54:34:
    6c:a7:0c:78:52:f3:2c:00:f8:9a:6b:41:69:00:89:21:29:94:
    50:55:b1:d6:9c:82:4b:aa:60:06:d4:35:6c:dc:b7:3c:93:cd:
    d8:f3:56:62:90:df:dc:33:59:d7:ab:eb:00:9e:1a:e1:2c:d7:
    2d:70:a8:38:dc:34:30:6c:59:71:2e:bf:85:cb:49:ef:95:73:
    f9:d4:cc:ac:0a:ca:c1:c9:32:b5:68:10:4b:fe:5e:73:fd:7f:
    c2:53:19:9f:8c:65:86:8f:84:70:83:cc:3d:f0:81:e6:15:b5:
    ab:31:94:d9:68:d4:6a:44:d5:c7:31:78:82:86:a6:9d:f1:6e:
    29:b1:ca:12:85:d5:3a:8f:42:dd:d1:10:f6:e9:f8:53:27:75:
    c4:7c:63:f1:6b:d0:72:4e:fd:22:7e:7f:4b:69:7d:85:da:a9:
    e1:eb:9d:e0:16:98:07:ab:4d:44:ac:65:52:7b:52:0a:f0:8d:
    ae:24:76:c2:b0:cd:b0:cb:90:e7:e2:89:45:e1:b9:d6:f3:34:
    3b:3c:ca:9a:54:9c:80:81:da:a9:b5:5e:34:56:48:f5:7e:93:
    a6:c3:b6:6c:b1:19:5f:6c:72:a8:ac:38:c6:ba:de:38:81:19:
    46:82:6b:99:7d:d5:43:fe:2e:f6:18:f5:e1:06:ed:ef:6f:e4:
    85:8a:3a:cf:f1:5c:91:ce:73:54:3f:9d:5b:87:e3:3f:6c:ed:
    55:34:8f:cc:1a:88:2b:7f:87:cc:08:1e:c1:85:ba:70:9e:80:
    6a:7f:06:4a:5d:c8:7d:1a:a3:e9:69:4e:b9:b5:a3:f0:6a:f5:
    53:47:4e:13:b4:a4:05:7e:b0:fd:7e:b5:ba:ab:07:ac:ec:a0:
    fc:4d:d9:d1:67:33:fd:5f:96:e6:26:71:6b:84:a9:9b:4c:78:
    b1:fa:4d:cd:ff:db:36:68:d3:7b:83:36:31:7c:a7:fe:15:f1:
    e4:c7:47:7f:f1:d9:76:13:ac:fe:5c:f0:29:41:8b:90:4c:1c:
    cb:7f:1d:ea:16:2d:52:3b

```

One-item revocation list

```
GET /puppet-ca/v1/certificate_revocation_list/ca
```

```
HTTP/1.1 200 OK
Content-Type: text/plain
```

```
-----BEGIN X509 CRL-----
MIICnDCBhQIBATANBgkqhkiG9w0BAQUFADAfMR0wGwYDVQQDDDBRQdXBwZXQgQ0E6
IGxvY2FsaG9zdBcNMTMxMDA3MTk0ODQwWWhcNMTgxMDA2MTk0ODQxWjAiMCACAQUX
DTEzMTAwNzE5NDg0MVowDDAKBgNVHRUEAwBAAaAOMAwWCGYDVR0UBAMCAQEWdQYJ
KoZIhvcNAQEFBQADggIBALrh49WNdmrJOPCRntDlnxC0bmqZgl8ZwTv7TO9VkmCG
Ksvo8zR2aTIOH9VUKqWrE0squhtFJXl8dxL4PRlRiLbmh07dp+NHdu8ejTQpoOTp
h69xbQFT3oHcIdn2cBGrLJQcZgXsiswT0KJ8nuw6eDO93yXDrGuSUDou99M99wTw
2nnlkUQKW9b0vUI7t2ADF5U8/DES+1IrvBq2IEHmg4+ekZRCxeJMuqdlRl3gymcJ
osSPbRgIjCcli6zD3aK4Nq5OMMpVLV/VVPwyQb4GwW4Wj5iyNAP8d/EAqtZ2lZHUi
nvuXmRtUWHJwfi40D5T2GQXxuUjB4pnh8cFq7f89iUvqoCwFo7nRIacrrweNFMYYD
GxVJVMfz4PkP66ckIPQ5Uuey92dg5p2w4b2cp8NstxMdgcc3KAF483ItKA8uIDuU
ldbzwlv2k5qUjoImueHwKolbLmPyYmvFp7hbnV+WpFbvGjyIfW3BMankDEv4ig0L
MCw6n2GKvlhSWM6Mrk8JalyYOFIsji0RoVCZsflinIRt28haldXVTPyNtct9mGAV
6az5W/nyixIPrrHubTx28zhmuHZx6y3hQMCLmuYOT+e7F/eFsYXVEjuJjxjr33uA
O/ii4EkTlslgzvonOtoBoGElzQAogrZI3HCwFYvU2whLKr9cwv5bpRkUfPCMq4n
-----END X509 CRL-----
```

```
> openssl crl -inform PEM -in lrevoked.crl -text -noout
Certificate Revocation List (CRL):
```

```
Version 2 (0x1)
Signature Algorithm: sha1WithRSAEncryption
Issuer: /CN=Puppet CA: localhost
Last Update: Oct  7 19:48:40 2013 GMT
Next Update: Oct  6 19:48:41 2018 GMT
CRL extensions:
    X509v3 CRL Number:
        1
```

```
Revoked Certificates:
```

```
Serial Number: 05
Revocation Date: Oct  7 19:48:41 2013 GMT
CRL entry extensions:
    X509v3 CRL Reason Code:
        Key Compromise
```

```
Signature Algorithm: sha1WithRSAEncryption
ba:e1:e3:d5:8d:76:6a:c9:38:f0:91:9e:d0:f5:9f:10:8e:6e:
6a:99:82:5f:19:c1:3b:fb:4c:ef:55:92:60:86:2a:cb:e8:f3:
34:76:69:32:0e:1f:d5:54:2a:a5:ab:13:4b:2a:ba:1b:45:25:
79:7c:77:12:f8:3d:1d:51:88:b6:e6:84:ee:dd:a7:e3:47:76:
ef:1e:8d:34:29:a0:e4:e9:87:af:71:6d:01:53:de:81:dc:21:
d9:f6:70:11:ab:2c:94:1c:66:05:ec:8a:cc:13:d0:a2:7c:9e:
ec:3a:78:33:bd:df:25:c3:ae:0b:92:51:da:2e:f7:d3:3d:f7:
04:f0:da:79:f5:91:44:0a:5b:d6:f4:bd:42:3b:b7:60:03:17:
95:3c:fc:31:12:fb:52:2b:bc:1a:b6:20:41:e6:83:8f:9e:91:
94:42:c5:e2:4c:ba:a7:75:47:5d:e0:ca:67:09:a2:c4:8f:6d:
18:08:8c:29:62:eb:30:f7:68:ae:0d:ab:93:8c:32:95:4b:57:
f5:55:3f:0c:90:6f:81:b0:5b:85:a3:e6:2c:8d:02:9f:1d:fc:
40:2a:b5:9d:b5:64:75:22:9e:fb:97:99:1b:54:58:72:70:7e:
2e:34:0f:94:f6:19:05:f1:b9:48:c1:e2:99:e1:f1:c1:6a:ed:
ff:3d:89:4b:ea:a0:2c:05:a3:b9:d1:21:a7:2b:af:07:8d:14:
c6:03:1b:15:49:54:c7:f3:e0:f9:0f:eb:a7:24:20:f4:39:52:
e7:b2:f7:67:60:e6:9d:b0:e1:bd:9c:a7:c3:6c:b7:13:1d:81:
c7:37:28:01:78:f3:72:2d:28:0f:2e:20:3b:94:d5:d6:f3:c3:
5b:f6:93:9a:94:8e:82:26:b9:e1:f0:2a:89:5b:2e:63:f2:62:
6b:c5:a7:b8:5b:9d:5f:96:a4:56:ef:1a:3c:88:7d:6d:c1:31:
a9:e4:0c:4b:f8:8a:0d:0b:30:2c:3a:9f:61:8a:bf:58:52:58:
ce:8c:ae:4f:09:6b:5c:98:38:52:ec:8c:8d:11:a1:50:99:b1:
fd:62:36:24:53:db:c8:5a:95:d5:d5:4c:fc:8d:b5:cb:7d:98:
60:2f:e9:ac:f9:5b:f9:f2:8b:12:0f:ae:b1:ee:6d:3c:76:f3:
38:66:b8:76:71:eb:2d:e1:40:c0:8b:9a:e6:0e:4f:e7:bb:17:
f7:85:b1:85:d5:12:3b:89:8f:18:eb:df:7b:80:3b:f8:a2:e0:
49:13:96:cd:60:ce:fa:27:3a:da:01:a0:61:25:cd:00:28:82:
```



```
b6:48:dc:75:c2:c0:56:2f:53:6c:21:2c:aa:fd:73:0b:f9:6e:
94:64:51:f3:c2:31:0e:27
```

No node name given

```
GET /puppet-ca/v1/certificate_revocation_list

HTTP/1.1 400 Bad Request
Content-Type: text/plain

No request key specified in /puppet-ca/v1/certificate_revocation_list
```

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](#)

Metrics API endpoints

v1 metrics

By default, Puppet Server enables two optional web APIs for [Java Management Extension \(JMX\)](#) metrics, namely [managed beans \(MBeans\)](#). For the newer Jolokia-based metrics API, see [v2 \(Jolokia\) metrics](#) on page 186.

The metrics v1 API was introduced in Puppet Enterprise 2016.4 and is now open sourced. It is still enabled but is deprecated.

Note: The metrics described here are returned only when passing the `level=debug` URL parameter, and the structure of the returned data might change, or the endpoint might be removed, in future versions.

GET /metrics/v1/mbeans

The GET /metrics/v1/mbeans endpoint lists available MBeans.

Response keys

- The key is the name of a valid MBean.
- The value is a URI to use when requesting that MBean's attributes.

POST /metrics/v1/mbeans

The POST /metrics/v1/mbeans endpoint retrieves requested MBean metrics.

Query parameters

The query doesn't require any parameters, but the request body must contain a JSON object whose values are metric names, or a JSON array of metric names, or a JSON string containing a single metric's name.

For a list of metric names, make a GET request to /metrics/v1/mbeans.

Response keys

The response format, though always JSON, depends on the request format:

- Requests with a JSON object return a JSON object where the values of the original object are transformed into the Mbeans' attributes for the metric names.
- Requests with a JSON array return a JSON array where the items of the original array are transformed into the Mbeans' attributes for the metric names.
- Requests with a JSON string return a JSON object of the Mbean's attributes for the given metric name.

GET `/metrics/v1/mbeans/<name>`

The GET `/metrics/v1/mbeans/<name>` endpoint reports on a single metric.

Query parameters

The query doesn't require any parameters, but the endpoint itself must correspond to one of the metrics returned by a GET request to `/metrics/v1/mbeans`.

Response keys

The endpoint's responses contain a JSON object mapping strings to values. The keys and values returned in the response vary based on the specified metric.

Example

Use `curl` from localhost to request data on MBean memory usage:

```
curl 'http://localhost:8080/metrics/v1/mbeans/java.lang:type=Memory'
```

The response should contain a JSON object representing the data:

```
{
  "ObjectPendingFinalizationCount" : 0,
  "HeapMemoryUsage" : {
    "committed" : 807403520,
    "init" : 268435456,
    "max" : 3817865216,
    "used" : 129257096
  },
  "NonHeapMemoryUsage" : {
    "committed" : 85590016,
    "init" : 24576000,
    "max" : 184549376,
    "used" : 85364904
  },
  "Verbose" : false,
  "ObjectName" : "java.lang:type=Memory"
}
```

v2 (Jolokia) metrics

By default, Puppet Server enables two optional web APIs for [Java Management Extension \(JMX\)](#) metrics, namely [managed beans \(MBeans\)](#). For the older metrics API, see [v1 metrics](#) on page 185.

Jolokia endpoints

The v2 metrics endpoint uses the [Jolokia](#) library, an extensive open-source metrics library with its own documentation.

The documentation below provides only the information you need to use the metrics as configured by default for Puppet Server, but Jolokia offers more features than are described below. Consult the [Jolokia documentation](#) for more information.

For security reasons, we enable only the read-access Jolokia interface by default:

- `read`
- `list`

- version
- search

Configuring Jolokia

To change the security access policy, create the `/etc/puppetlabs/puppetserver/jolokia-access.xml` file with contents that follow the [Jolokia access policy](#) and uncomment the `metrics.metrics-webservice.jolokia.servlet-init-params.policyLocation` parameter before restarting puppetserver.

The `metrics.metrics-webservice.jolokia.servlet-init-params` table within the [metrics.conf](#) on page 116 file provides more configuration options. See Jolokia's [agent initialization documentation](#) for all of the available options.

Disabling the endpoints

To disable the v2 endpoints, set the `metrics.metrics-webservice.jolokia.enabled` parameter in `metrics.conf` to `false`.

Usage

You can query the metrics v2 API using GET or POST requests.

GET /metrics/v2/

(Introduced in Puppet Server 5)

This endpoint requires an operation, and depending on the operation can accept or might require an additional query:

```
GET /metrics/v2/<OPERATION>/<QUERY>
```

Response

A successful request returns a JSON document.

Examples

To list all valid mbeans querying the metrics endpoint

```
GET /metrics/v2/list
```

Which should return a response similar to

```
{
  "request": {
    "type": "list"
  },
  "value": {
    "java.util.logging": {
      "type=Logging": {
        "op": {
          "getLoggerLevel": {
            ...
          },
          ...
        },
        "attr": {
          "LoggerNames": {
            "rw": false,
            "type": "[Ljava.lang.String;",
            "desc": "LoggerNames"
          },
          "ObjectName": {
            "rw": false,
            "type": "javax.management.ObjectName",

```

```

        "desc": "ObjectName"
      },
    },
    "desc": "Information on the management interface of the MBean"
  },
  ...
}
}

```

So, from the example above we could query for the registered logger names with this HTTP call:

```
GET /metrics/v2/read/java.util.logging:type=Logging/LoggerNames
```

Which would return the JSON document

```

{
  "request": {
    "mbean": "java.util.logging:type=Logging",
    "attribute": "LoggerNames",
    "type": "read"
  },
  "value": [
    "javax.management.snmp",
    "global",
    "javax.management.notification",
    "javax.management.modelmbean",
    "javax.management.timer",
    "javax.management",
    "javax.management.mlet",
    "javax.management.mbeanserver",
    "javax.management.snmp.daemon",
    "javax.management.relation",
    "javax.management.monitor",
    "javax.management.misc",
    ""
  ],
  "timestamp": 1497977258,
  "status": 200
}

```

The MBean names can then be created by joining the the first two keys of the value table with a colon (the domain and prop list in Jolokia parlance). Querying the MBeans is achieved via the read operation. The read operation has as its GET signature:

```
GET /metrics/v2/read/<MBEAN NAMES>/<ATTRIBUTES>/<OPTIONAL INNER PATH FILTER>
```

```
POST /metrics/v2/<OPERATION>
```

You can also submit a POST request with the query as a JSON document in the body of the POST.

Filtering

The new Jolokia-based metrics API also provides globbing (wildcard selection) and response filtering features.

Example

You can combine both of these features to query garbage collection data, but return only the collection counts and times.

```
GET metrics/v2/read/java.lang:name=*,type=GarbageCollector/
CollectionCount,CollectionTime
```

This returns a JSON response:

```
{
  "request": {
    "mbean": "java.lang:name=*,type=GarbageCollector",
    "attribute": [
      "CollectionCount",
      "CollectionTime"
    ],
    "type": "read"
  },
  "value": {
    "java.lang:name=PS Scavenge,type=GarbageCollector": {
      "CollectionTime": 1314,
      "CollectionCount": 27
    },
    "java.lang:name=PS MarkSweep,type=GarbageCollector": {
      "CollectionTime": 580,
      "CollectionCount": 5
    }
  },
  "timestamp": 1497977710,
  "status": 200
}
```

Refer to the [Jolokia protocol documentation](#) for more advanced usage.

Status API endpoints

Services endpoint

The `services` endpoint of Puppet Server's Status API provides information about services running on Puppet Server. As of Puppet Server 2.6.0, the endpoint provides information about memory usage similar to the data produced by the Java MemoryMXBean, as well as basic data on the `puppetserver` process's state and uptime. See the [Java MemoryMXBean documentation](#) for help interpreting the memory information.

Note: This is an experimental feature provided for troubleshooting purposes. In future releases, the `services` endpoint's response payload might change without warning.

For information about HTTP client metrics, which are served from the status endpoint, see [HTTP Client Metrics](#) on page 151.

GET `/status/v1/services`

(Introduced in Puppet Server 2.6.0)

Supported HTTP methods

GET

Supported formats

JSON

Query parameters

- `level` (optional): The response includes status information for all registered services at the requested level of detail. Default: `info`. Valid values:
 - `critical`: Returns the minimum amount of status information for each service. This level returns data quickly and is suitable for frequently updating uses, such as health checks for a load balancer.
 - `info`: Returns more info than the `critical` level for each service. The specific data depends on the implementation details of the services loaded in the application, but generally includes enough human-readable data to provide a quick impression of each service's health and status.
 - `debug`: This level returns status information about a service in enough detail to be suitable for debugging issues with the `puppetserver` process. Depending on the service, this level can be significantly more expensive than lower levels, reduce the process's performance, and generate large amounts of data. This level is suitable for producing aggregate metrics about the performance or resource usage of Puppet Server's subsystems.

The information returned for any service at each increasing level of detail includes the data from lower levels. In other words, the `info` level returns the same data structure as the `critical` level, and might provide additional data in the `status` field depending on the service. Likewise, the `debug` level returns the same data structure as `info`, and might also add additional information in the `status` field.

Response

The `services` endpoint's response includes information for each service about which the Status service is aware. Each service's `state` value is one of the following:

- `running`, if and only if all services are running
- `error` if any service reports an error
- `starting` if any service reports that it is starting, and no service reports an error or that it is stopping
- `stopping` if any service reports that it is stopping and no service reports an error
- `unknown` if any service reports an unknown state and no services report an error

Requests to this endpoint return one of the following status codes:

- 200 when all services are in running state.
- 404 when a requested service is not found.
- 503 when the service state is unknown, error, starting, or stopping

Example request and response for a debug-level GET request

```
GET /status/v1/services?level=debug

HTTP/1.1 200 OK
Content-Type: application/json

{
  "status-service": {
    "detail_level": "debug",
    "service_status_version": 1,
    "service_version": "0.3.5",
    "state": "running",
    "status": {
      "experimental": {
        "jvm-metrics": {
          "heap-memory": {
            "committed": 1049100288,
            "init": 268435456,
            "max": 1908932608,
            "used": 216512656
          },
          "non-heap-memory": {
            "committed": 256466944,
```

```

        "init": 2555904,
        "max": -1,
        "used": 173201432
      },
      "start-time-ms": 1472496731281,
      "up-time-ms": 538974
    }
  }
}

```

Authorization

Requests to the `services` endpoint are authorized by the [auth.conf](#) on page 78 as of Puppet Server 5.3.0. For more information about the supported Puppet Server authorization processes and configuration settings, see the [auth.conf](#) on page 78.

One may also restrict access to the status service by changing the `client-auth` setting to `required` for the webserver. See [Configuring the Webserver Service](#) for more information on the `client-auth` setting.

Simple endpoint

The `simple` endpoint of Puppet Server's Status API provides a simple indication of whether Puppet Server is running on a server. It's designed for load balancers that don't support any kind of JSON parsing or parameter setting and returns a simple string body (either the state of the server or a simple error message) and a status code relevant to the result.

The content type for this endpoint is `text/plain; charset=utf-8`.

GET /status/v1/simple

(Introduced in Puppet Server 2.6.0)

Supported HTTP methods

GET

Supported formats

Plain text

Query parameters

None

Response

The `simple` endpoint's response consists of a single word describing Puppet Server's status:

- `running`, if and only if the Puppet Server service is running
- `error`, if the service reports an error
- `unknown`, if the service reports an unknown state, but doesn't report an error

Requests to this endpoint return one of the following status codes:

- 200 if and only if the Puppet Server service reports a status of running
- 503 if the service's status is unknown or error

Example request and response for a GET request

```

GET /status/v1/simple

HTTP/1.1 200 OK
Content-Type: application/json

```

```
running
```

Authorization

Requests to the `simple` endpoint are authorized by the [auth.conf](#) on page 78 as of Puppet Server 5.3.0. For more information about the supported Puppet Server authorization processes and configuration settings, see the [auth.conf](#) on page 78.

One may also restrict access to the status service by changing the `client-auth` setting to `required` for the webserver. See [Configuring the Webserver Service](#) for more information on the `client-auth` setting.

Certificate Clean

The `certificate clean` endpoint of the CA API allows you to revoke and delete a list of certificates with a single request.

```
PUT /puppet-ca/v1/clean
Content-Type: application/json
```

The request body takes one required key — `certnames`. This includes the list of certificates for the endpoint to clean. Each certificate in the list is revoked, and the associated certificate file deleted from the CA.

If a certname does not have an associated signed cert on the CA, the response body calls this out, but the request does not error.

Example

In the following example, both certs are revoked and their files deleted.

```
PUT /puppet-ca/v1/clean
Content-Type: application/json
Content-Length: 58

{"certnames":["agent1.example.net","agent2.example.net"]}

HTTP/1.1 200 OK
Content-Type: text/plain
Successfully cleaned all certificates.
```

In the following example, the missing certificate is skipped, and the other is revoked and deleted.

```
PUT /puppet-ca/v1/clean
Content-Type: application/json
Content-Length: 58

{"certnames":["missing.example.net","agent1.example.net"]}

HTTP/1.1 200 OK
Content-Type: text/plain
The following certs do not exist and cannot be revoked:
["missing.example.net"]
```

Server-specific Puppet API endpoints

Environment classes

The environment classes API serves as a replacement for the Puppet resource type API for [classes](#), which was removed in Puppet 5.

Changes in the environment classes API

Compared to the resource type API, the environment classes API covers different things, returns new or different information, and omits some information.

Covers classes only

The environment classes API covers only [classes](#), whereas the resource type API covers classes, [node definitions](#), and [defined types](#).

Changes class information caching behavior

Queries to the resource type API use cached class information per the configuration of the [environment_timeout](#) setting, as set in the corresponding environment's `environment.conf` file. The environment classes API does not use the value of `environment_timeout` with respect to the data that it caches. Instead, only when the `environment-class-cache-enabled` setting in the `jrubby-puppet` configuration section is set to `true`, the environment classes API uses HTTP [Etags](#) to represent specific versions of the class information. And it uses the Puppet Server [Environment cache](#) on page 202 as an explicit mechanism for marking an Etag as expired. See the [Headers and caching behavior](#) section for more information about caching and invalidation of entries.

Uses typed values

The environment classes API includes a `type`, if defined for a class parameter. For example, if the class parameter were defined as `String $some_str`, the `type` parameter would hold a value of `String`.

Provides default literal values

For values that can be presented in pure JSON, the environment classes API provides a `default_literal` form of a class parameter's default value. For example, if an `Integer` type class parameter were defined in the manifest as having a default value of 3, the `default_literal` element for the parameter will contain a JSON Number type of 3.

Lacks filters

The environment classes API does not provide a way to filter the list of classes returned via use of a search string. The environment classes API returns information for all classes found within an environment's manifest files.

Includes filenames

Unlike the resource type API in Puppet 4, the environment classes API does include the filename in which each class was found. The resource type API in Puppet 3 does include the filename, but the resource type API under Puppet 4 does not.

Lacks line numbers

The environment classes API does not include the line number at which a class is found in the file.

Lacks documentation strings (vs. Puppet 3)

Unlike the resource type API in Puppet 3, the environment classes API does not include any doc strings for a class entry. Note that doc strings are also not returned for class entries in the Puppet 4 resource type API.

Returns file entries for manifests with no classes

The environment classes API returns a file entry for manifests that exist in the environment but in which no classes were found. The resource type API omits entries for files which do not contain any classes.

Uses application/json Content-Type

The Content-Type in the response to an environment classes API query is `application/json`, whereas the resource type API uses a Content-Type of `text/pson`.

Includes successfully parsed classes, even if some return errors, and returns error messages

The environment classes API includes information for every class that can successfully be parsed. For any errors which occur when parsing individual manifest files, the response includes an entry for the corresponding manifest file, along with an error and detail string about the failure.

In comparison, if an error is encountered when parsing a manifest, the resource type API omits information from the manifest entirely. It includes class information from other manifests that it successfully parsed, assuming none of the parsing errors were found in one of the files associated with the environment's [manifest setting](#). If one or more classes is returned but errors were encountered parsing other manifests, the response from the resource type API call doesn't include any explicit indication that a parsing error was encountered.

GET /puppet/v3/environment_classes?environment=:environment

(Introduced in Puppet Server 2.3.0.)

Making a request with no query parameters is not supported and returns an HTTP 400 (Bad Request) response.

Supported HTTP Methods

GET

Supported Formats

JSON

Query Parameters

Provide one parameter to the GET request:

- **environment:** Only the classes and parameter information pertaining to the specified environment will be returned for the call.

Responses

GET request with results

```
GET /puppet/v3/environment_classes?environment=env

HTTP/1.1 200 OK
Etag: b02ede6ecc432b134217a1cc681c406288ef9224
Content-Type: application/json

{
  "files": [
    {
      "path": "/etc/puppetlabs/code/environments/env/manifests/site.pp",
      "classes": []
    },
    {
      "path": "/etc/puppetlabs/code/environments/env/modules/mymodule/manifests/init.pp",
      "classes": [
        {
          "name": "mymodule",
          "params": [
            {
              "default_literal": "this is a string",
              "default_source": "\"this is a string\"",
              "name": "a_string",
              "type": "String"
            },
            {
              "default_literal": 3,
              "default_source": "3",
              "name": "an_integer",
              "type": "Integer"
            }
          ]
        }
      ]
    }
  ]
}
```

```

      "error": "Syntax error at '=>' at /etc/puppetlabs/code/environments/
env/modules/mymodule/manifests/other.pp:20:19",
      "path": "/etc/puppetlabs/code/environments/env/modules/mymodule/
manifests/other.pp"
    }
  ],
  "name": "env"
}

```

GET request with Etag roundtripped from a previous GET request

If you send the [Etag](#) value that was returned from the previous request to the server in a follow-up request, and the underlying environment cache has not been invalidated, the server will return an HTTP 304 (Not Modified) response. See the [Headers and Caching Behavior](#) section for more information about caching and invalidation of entries.

```

GET /puppet/v3/environment_classes?environment=env
If-None-Match: b02ede6ecc432b134217a1cc681c406288ef9224

HTTP/1.1 304 Not Modified
Etag: b02ede6ecc432b134217a1cc681c406288ef9224

```

If the environment cache has been updated from what was used to calculate the original Etag, the server will return a response with the full set of environment class information:

```

GET /puppet/v3/environment_classes?environment=env
If-None-Match: b02ede6ecc432b134217a1cc681c406288ef9224

HTTP/1.1 200 OK
Etag: 2f4f83096265b9741c5304b3055f866df0336762
Content-Type: application/json

{
  "files": [
    {
      "path": "/etc/puppetlabs/code/environments/env/manifests/site.pp",
      "classes": []
    },
    {
      "path": "/etc/puppetlabs/code/environments/env/modules/mymodule/
manifests/init.pp",
      "classes": [
        {
          "name": "mymodule",
          "params": [
            {
              "default_literal": "this is a string",
              "default_source": "\"this is a string\"",
              "name": "a_string",
              "type": "String"
            },
            {
              "default_literal": 3,
              "default_source": "3",
              "name": "an_integer",
              "type": "Integer"
            },
            {
              "default_literal": {
                "one": "foo",
                "two": "hello"
              },
              "default_source": "{ \"one\" => \"foo\", \"two\" => \"hello
\" }",

```

```

        "name": "a_hash",
        "type": "Hash"
      }
    ]
  }
]
},
"env": "env"
}

```

Environment does not exist

If you send a request with an environment parameter that doesn't correspond to the name of a directory environment on the server, the server returns an HTTP 404 (Not Found) error:

```

GET /puppet/v3/environment_classes?environment=doesnotexist

HTTP/1.1 404 Not Found

Could not find environment 'doesnotexist'

```

No environment given

```

GET /puppet/v3/environment_classes

HTTP/1.1 400 Bad Request

You must specify an environment parameter.

```

Environment parameter specified with no value

```

GET /puppet/v3/environment_classes?environment=

HTTP/1.1 400 Bad Request

The environment must be purely alphanumeric, not ''

```

Environment includes non-alphanumeric characters

If the environment parameter in your request includes any characters that are not A-Z, a-z, 0-9, or _ (underscore), the server returns an HTTP 400 (Bad Request) error:

```

GET /puppet/v3/environment_classes?environment=bog|us

HTTP/1.1 400 Bad Request

The environment must be purely alphanumeric, not 'bog|us'

```

Schema

An environment classes response body conforms to the [environment classes schema](#).

Headers and Caching Behavior

If the `environment-class-cache-enabled` setting in the `jrubby-puppet` configuration section is set to `true`, the environment classes API caches the response data. This can provide a significant performance benefit by reducing the amount of data that needs to be provided in a response when the underlying Puppet code on disk remains unchanged from one request to the next. Use of the cache does, however, require that cache entries are invalidated after Puppet code has been updated.

To avoid invalidated cache entries, you can omit the `environment-class-cache-enabled` setting from a node's configuration or set it to `false`. In this case, the server discovers and parses manifests for every incoming

request. This can significantly increase bandwidth overhead for repeated requests, particularly when there are few changes to the underlying Puppet code. However, this approach ensures that the latest available data is returned to every request.

Behaviors when the environment class cache is enabled

When the `environment-class-cache-enabled` setting is set to `true`, the response to a query to the `environment_classes` endpoint includes an HTTP [Etag](#) header. The value for the Etag header is a hash that represents the state of the latest class information available for the requested environment. For example:

```
Etag: 31d64b8038258202b4f5eb508d7dab79c46327bb
```

A client can (but is not required to) provide the Etag value back to the server in a subsequent `environment_classes` request. The client would provide the tag value as the value for an [If-None-Match](#) HTTP header:

```
If-None-Match: 31d64b8038258202b4f5eb508d7dab79c46327bb
```

If the latest state of code available on the server matches that of the value in the `If-None-Match` header, the server returns an HTTP 304 (Not Modified) response with no response body. If the server has newer code available than what is captured by the `If-None-Match` header value, or if no `If-None-Match` header is provided in the request, the server parses manifests again. Assuming the resulting payload is different than a previous request's, the server provides a different Etag value and new class information in the response payload.

If the client sends an `Accept-Encoding: gzip` HTTP header for the request and the server provides a gzip-encoded response body, the server might append the characters `--gzip` to the end of the Etag. For example, the HTTP response headers could include:

```
Content-Encoding: gzip
Etag: e84bbce5482243b3eb3a190e5c90e535cf4f20de--gzip
```

The server accepts both forms of an Etag (with or without the trailing `--gzip` characters) as the same value when validating it in a request's `If-None-Match` header against its cache.

It is best, however, for clients to use the Etag without parsing its content. A client expecting an HTTP 304 (Not Modified) response if the cache has not been updated since the prior request should provide the exact value returned in the Etag header from one request, to the server in an `If-None-Match` header in a subsequent request for the environment's class information.

Clearing class information cache entries

After updating an environment's manifests, you must clear the server's class information cache entries, so the server can parse the latest manifests and reflect class changes to the class information in queries to the environment classes endpoint. To clear cache entries on the server, do one of the following:

- Call the [Environment cache](#) on page 202.

For best performance, call this endpoint with a query parameter that specifies the environment whose cache should be flushed.

- Restart Puppet Server.

Each environment's cache is held in memory for the Puppet Server process and is effectively flushed whenever Puppet Server is restarted, whether with a [Restarting Puppet Server](#) on page 161 or a full JVM restart.

Authorization

All requests made to the environment classes API are authorized using the Trapperkeeper-based [auth.conf](#) on page 78.

For more information about the Puppet Server authorization process and configuration settings, see the [auth.conf](#) on page 78.

Environment modules

The environment modules API will return information about what modules are installed for the requested environment.

GET /puppet/v3/environment_modules

Supported HTTP Methods

GET

Supported Formats

JSON

Responses

GET request with results

```
GET /puppet/v3/environment_modules

HTTP/1.1 200 OK
Content-Type: application/json

[ {
  "modules": [
    {
      "name": "puppetlabs/ntp",
      "version": "6.0.0"
    },
    {
      "name": "puppetlabs/stdlib",
      "version": "4.14.0"
    }
  ],
  "name": "env"
},
{
  "modules": [
    {
      "name": "puppetlabs/stdlib",
      "version": "4.14.0"
    },
    {
      "name": "puppetlabs/azure",
      "version": "1.1.0"
    }
  ],
  "name": "production"
} ]
```

GET /puppet/v3/environment_modules?environment=:environment

Supported HTTP Methods

GET

Supported Formats

JSON

Query Parameters

Provide one parameter to the GET request:

- `environment`: Request information about modules pertaining to the specified environment only.

Responses

GET request with results

```
GET /puppet/v3/environment_modules?environment=env

HTTP/1.1 200 OK
Content-Type: application/json

{
  "modules": [
    {
      "name": "puppetlabs/ntp",
      "version": "6.0.0"
    },
    {
      "name": "puppetlabs/stdlib",
      "version": "4.14.0"
    }
  ],
  "name": "env"
}
```

Environment does not exist

If you send a request with an environment parameter that doesn't correspond to the name of a directory environment on the server, the server returns an HTTP 404 (Not Found) error:

```
GET /puppet/v3/environment_modules?environment=doesnotexist

HTTP/1.1 404 Not Found

Could not find environment 'doesnotexist'
```

No environment given

```
GET /puppet/v3/environment_modules

HTTP/1.1 400 Bad Request

An environment parameter must be specified
```

Environment parameter specified with no value

```
GET /puppet/v3/environment_modules?environment=

HTTP/1.1 400 Bad Request

The environment must be purely alphanumeric, not ''
```

Environment includes non-alphanumeric characters

If the environment parameter in your request includes any characters that are not A-Z, a-z, 0-9, or _ (underscore), the server returns an HTTP 400 (Bad Request) error:

```
GET /puppet/v3/environment_modules?environment=bog|us

HTTP/1.1 400 Bad Request

The environment must be purely alphanumeric, not 'bog|us'
```

No metadata.json file

If your modules do not have a [metadata.json](#) file, puppetserver will not be able to determine the version of your module. In this case, puppetserver will return a null value for `version` in the response body.

Schema

An environment modules response body conforms to the [environment modules schema](#).

Validating your json

If you have a response body that you'd like to validate against the [environment_modules.json](#) schema, you can do so using the ruby library [json-schema](#).

First, install the ruby gem to be used:

```
gem install json-schema
```

Next, given a json file, you can validate its schema.

Here is a basic json file called *example.json*:

```
{
  "modules": [
    {
      "name": "puppetlabs/ntp",
      "version": "6.0.0"
    },
    {
      "name": "puppetlabs/stdlib",
      "version": "4.16.0"
    }
  ],
  "name": "production"
}
```

Run this command from the root dir of the puppetserver project (or update the path to the json schema file in the command below):

```
ruby -rjson-schema -e "puts JSON::Validator.validate!('./documentation/
puppet-api/v3/environment_modules.json','example.json')"
```

If the json is a valid schema, the command should output `true`. Otherwise, the library will print a schema validation error detailing which key or keys validate the schema.

If you have a response that is the entire list of environment modules (i.e. the `environment_modules` endpoint), you will need to use this command to validate the json schema:

```
ruby -rjson-schema -e "puts JSON::Validator.validate!('./documentation/
puppet-api/v3/environment_modules.json','all.json', :list=>true)"
```

Authorization

All requests made to the environment modules API are authorized using the Trapperkeeper-based [auth.conf](#) on page 78.

For more information about the Puppet Server authorization process and configuration settings, see the [auth.conf](#) on page 78.

Static file content

The `static_file_content` endpoint returns the standard output of a [code-content-command](#) script, which should output the contents of a specific version of a [file resource](#) that has a source attribute with a `puppet:///` URI value. That source must be a file from the `files` or `tasks` directory of a module in a specific [environment](#).

Puppet Agent uses this endpoint only when applying a [static catalog](#).

GET /puppet/v3/static_file_content/<FILE-PATH>

(Introduced in Puppet Server 2.3.0)

To retrieve a specific version of a file at a given environment and path, make an HTTP request to this endpoint with the required parameters.

The <FILE-PATH> segment of the endpoint is required. The path corresponds to the requested file's path on the Server relative to the given environment's root directory, and must point to a file in the `*/*/files/**`, `*/*/lib/**`, or `*/*/tasks/**` glob. For example, Puppet Server will inline metadata into static catalogs for file resources sourcing module files located by default in `/etc/puppetlabs/code/environments/<ENVIRONMENT>/modules/<MODULE NAME>/files/**`.

Query parameters

You must also pass two parameters in the GET request:

- `code_id`: a unique string provided by the [catalog](#) that identifies which version of the file to return.
- `environment`: the environment that contains the desired file.

Response

A successful request to this endpoint returns an HTTP 200 response code and `application/octet-stream` Content-Type header, and the contents of the specified file's requested version in the response body. An unsuccessful request returns an error response code with a `text/plain` Content-Type header:

- 400: returned when any of the parameters are not provided.
- 403: returned when requesting a file that is not within a module's `files` or `tasks` directory.
- 500: returned when `code-content-command` is not configured on the server, or when a requested file or version is not present in a repository.

Example response

Consider a server `localhost`, with a versioned file located at `/modules/example/files/data.txt` in the `production` environment. The version is identified by a `code_id` of `urn:puppet:code-id:1:67eb71417fbd736a619c8b5f9bfc0056ea8c53ca;production`, and that version of the file contains `Puppet test`.

If you run this command:

```
curl -i -k 'https://localhost:8140/puppet/v3/static_file_content/
modules/example/files/data.txt?code_id=urn:puppet:code-
id:1:67eb71417fbd736a619c8b5f9bfc0056ea8c53ca;production&environment=production'
```

Puppet Server returns:

```
HTTP/1.1 200 OK
Date: Wed, 2 Mar 2016 23:44:08 GMT
X-Puppet-Version: 4.4.0
Content-Length: 4
Server: Jetty(9.2.10.v20150310)

Puppet test
```

Notes

When requesting a file from this endpoint, Puppet Server passes the values of the `file-path`, `code_id`, and `environment` parameters as arguments to the `code-content-command` script. If the script returns an exit code of 0, Puppet Server returns the script's standard output, which should be the contents of the requested version of the file.

This endpoint returns an error (status 500) if the `code-content-command` setting is not configured on Puppet Server.

Note: The `code-content-command` and `code-id-command` scripts are not provided in a default installation or upgrade. For more information about these scripts, see the [static catalog documentation](#).

Authorization

All requests made to the static file content API are authorized using the Trapperkeeper-based [auth.conf](#).

For more information about the Puppet Server authorization process and configuration settings, see the [auth.conf documentation](#).

Administrative API endpoints

Environment cache

When using directory environments, Puppet Server [caches](#) the data it loads from disk for each environment. It provides an endpoint for clearing this cache:

DELETE `/puppet-admin-api/v1/environment-cache`

To trigger a complete invalidation of the data in this cache, make an HTTP request to this endpoint.

Query Parameters

(Introduced in Puppet Server 1.1/2.1)

This endpoint accepts an optional query parameter, `environment`, whose value may be set to the name of a specific Puppet environment. If this parameter is provided, only the specified environment will be flushed from the cache, as opposed to all environments.

Response

A successful request to this endpoint will return an HTTP 204: No Content. The response body will be empty.

Example

```
$ curl -i --cert <PATH TO CERT> --key <PATH TO KEY> --cacert <PATH TO PUPPET
  CA CERT> -X DELETE https://localhost:8140/puppet-admin-api/v1/environment-
  cache
HTTP/1.1 204 No Content

$ curl -i --cert <PATH TO CERT> --key <PATH TO KEY> --cacert <PATH TO PUPPET
  CA CERT> -X DELETE https://localhost:8140/puppet-admin-api/v1/environment-
  cache?environment=production
HTTP/1.1 204 No Content
```

Relevant Configuration

Access to this endpoint is controlled via Puppet Server's [auth.conf](#) file.

JRuby pool

Puppet Server contains a pool of JRuby instances, and provides an API endpoint for flushing it:

DELETE `/puppet-admin-api/v1/jruby-pool`

This will remove all of the existing JRuby interpreters from the pool, allowing the memory occupied by these interpreters to be reclaimed by the JVM's garbage collector. The pool will then be refilled with new JRuby instances, each of which will load the latest Ruby code and related resources from disk.

If you're developing new Ruby plugins that run in Puppet Server (functions, resource types, report handlers), you may need to force Puppet to re-load its plugins when a new version is ready to test. Killing the JRuby instances will do this, and it's faster than restarting the entire JVM process.

Furthermore, if you are using multiple environments, this could be useful if you want to make sure that your JRuby instances are cleaned up and don't have conflicts based on common code that appears in multiple environments.

Note that this operation is computationally expensive, and as such Puppet Server will be unable to fulfill any incoming requests until the first of the new interpreters has been initialized, which may take several seconds.

Response

A successful request to this endpoint will return an HTTP 204: No Content. The response body will be empty.

Example

```
$ curl -i --cert <PATH TO CERT> --key <PATH TO KEY> --cacert <PATH TO PUPPET
  CA CERT> -X DELETE https://localhost:8140/puppet-admin-api/v1/jruby-pool
HTTP/1.1 204 No Content
```

GET /puppet-admin-api/v1/jruby-pool/thread-dump

Retrieve a Ruby thread dump for each JRuby instance registered to the pool. The thread dump provides a backtrace through the Ruby code that each instance is executing and is useful for diagnosing instances that have stalled or are otherwise unresponsive. Backtraces are generated using the JRuby JMX interface and require the `jruby.management.enabled` property to be set to `true` in the JVM running Puppet Server.

Response

A successful request to this endpoint will return a HTTP 200: Ok status code. The response body will be a JSON document containing a map that associates each JRuby instance ID with a map containing a `thread-dump` entry that has a string value with the Ruby backtrace.

A HTTP 500: Internal Server Error status code will be returned if an exception occurs while retrieving the thread dump for a JRuby instance, or if the `jruby.management.enabled` property is not set to `true`. The response body in this case is also JSON, but the failed instances will be associated with a map containing an `error` entry with a value describing the issue.

Example

```
$ curl -si --cert <PATH TO CERT> --key <PATH TO KEY> --cacert <PATH TO
  PUPPET CA CERT> -X GET https://localhost:8140/puppet-admin-api/v1/jruby-
  pool/thread-dump
HTTP/1.1 200 OK

{"1":{"thread-dump":"All threads known to Ruby instance 1960016402\n
\n ..."}}

# Error returned when jruby.management.enabled is not configured
$ curl -si --cert <PATH TO CERT> --key <PATH TO KEY> --cacert <PATH TO
  PUPPET CA CERT> -X GET https://localhost:8140/puppet-admin-api/v1/jruby-
  pool/thread-dump
HTTP/1.1 500 Server Error

{"1":{"error":"JRuby management interface not enabled. Add '-
  Djruby.management.enabled=true' to JAVA_ARGS to enable thread dumps."}}
```

Relevant Configuration

Access to this endpoint is controlled via Puppet Server's [auth.conf](#) file.

Certificate authority and SSL

Puppet can use its built-in certificate authority (CA) and public key infrastructure (PKI) tools or use an existing external CA for all of its secure socket layer (SSL) communications.

Puppet uses certificates to verify the the identity of nodes. These certificates are issued by the certificate authority (CA) service of a Puppet primary server. When a node checks into the Puppet v for the first time, it requests a

certificate. The Puppet primary server examines this request, and if it seems safe, creates a certificate for the node. When the agent node picks up this certificate, it knows it can trust the Puppet primary server, and it can now identify itself later when requesting a catalog.

After installing the Puppet Server, before starting it for the first time, use the `puppetserver ca setup` command to create a default intermediate CA. For more complex use cases, see the [Intermediate and External CA documentation](#).

Note: For backward compatibility, starting Puppet Server before running `puppetserver ca setup` creates the old single-cert CA. This configuration is not recommended, so if you are using Puppet 6, use the `setup` command instead.

Puppet provides two command line tools for performing SSL tasks:

- `puppetserver ca` signs certificate requests and revokes certificates.
- `puppet ssl` performs agent-side tasks, such as submitting a certificate request or downloading a node certificate.

What's changed in Puppet 6

Puppet 6 removes the `puppet cert` command and its associated certificate-related faces. In Puppet 6 you must use the new subcommands listed above instead.

Puppet 6 also introduces full support for [intermediate CAs](#), the recommended architecture. This requires changes on both the server and the agent, so using it requires both the server and the agent to be updated to Puppet 6.

- [Puppet Server CA commands](#) on page 204

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

- [Intermediate CA](#) on page 140
- [Autosigning certificate requests](#) on page 208

Before Puppet agent nodes can retrieve their configuration catalogs, they require a signed certificate from the local Puppet certificate authority (CA). When using Puppet's built-in CA instead of an external CA, agents submit a certificate signing request (CSR) to the CA to retrieve a signed certificate after it's available.

- [CSR attributes and certificate extensions](#) on page 211

When Puppet agent nodes request their certificates, the certificate signing request (CSR) usually contains only their certname and the necessary cryptographic information. Agents can also embed additional data in their CSR, useful for policy-based autosigning and for adding new trusted facts.

- [Regenerating certificates in a Puppet deployment](#) on page 216

In some cases, you might need to regenerate the certificates and security credentials (private and public keys) that are generated by Puppet's built-in PKI systems.

- [External CA](#) on page 219

This information describes the supported and tested configurations for external CAs in this version of Puppet. If you have an external CA use case that isn't listed here, contact Puppet so we can learn more about it.

- [External SSL termination](#) on page 142

Puppet Server CA commands

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

CA subcommands

See [Subcommands](#) on page 134 to view the subcommands for the `puppetserver ca` command.

Some actions have additional options. Run `puppetserver ca help` for details.

The `puppetserver-ca` CLI tool is shipped as a gem alongside Puppet Server. It can be updated between releases to pick up improvements and bug fixes. To update the gem, run:

```
/opt/puppetlabs/puppet/bin/gem install -i /opt/puppetlabs/puppet/lib/ruby/vendor_gems puppetserver-ca
```

API authentication

For security, access to the `certificate_status` API endpoint that is issued to sign and revoke certificates is tightly restricted. In Puppet 6, the primary server's certificate is generated with a special extension which is added to the allowlist in the `auth.conf` entries for the `certificate_status` and `certificate_statuses` endpoint. This extension is reserved, and Puppet Server refuses to sign other CSRs requesting it, even with `allow-authorization-extensions` set to `true`. If you need a certificate with this extension, you can generate it offline by stopping Puppet Server and running `puppetserver ca generate --ca-client --certname <name>`. API authentication is required for regenerating the primary server cert. For details on cert regeneration, see [Regenerating certificates in a Puppet deployment](#).

Upgrading

The Puppet CA commands are available in Puppet 5, but to use them, you must update Puppet Server's `auth.conf` to include a rule allowing the primary server's certname to access the `certificate_status` and `certificate_statuses` endpoints. If you're upgrading from Puppet 5 to Puppet 6 and you are not regenerating your CA, you must allow the primary server's certname in your `auth.conf` file. See [Puppet Server configuration files: auth.conf](#) for details on how to use `auth.conf`.

The following example shows how to allow the CA commands to access the `certificate_status` endpoint:

```
{
  match-request: {
    path: "/puppet-ca/v1/certificate_status"
    type: path
    method: [get, put, delete]
  }
  allow: primaryserver.example.com
  sort-order: 500
  name: "puppetlabs cert status"
},
```

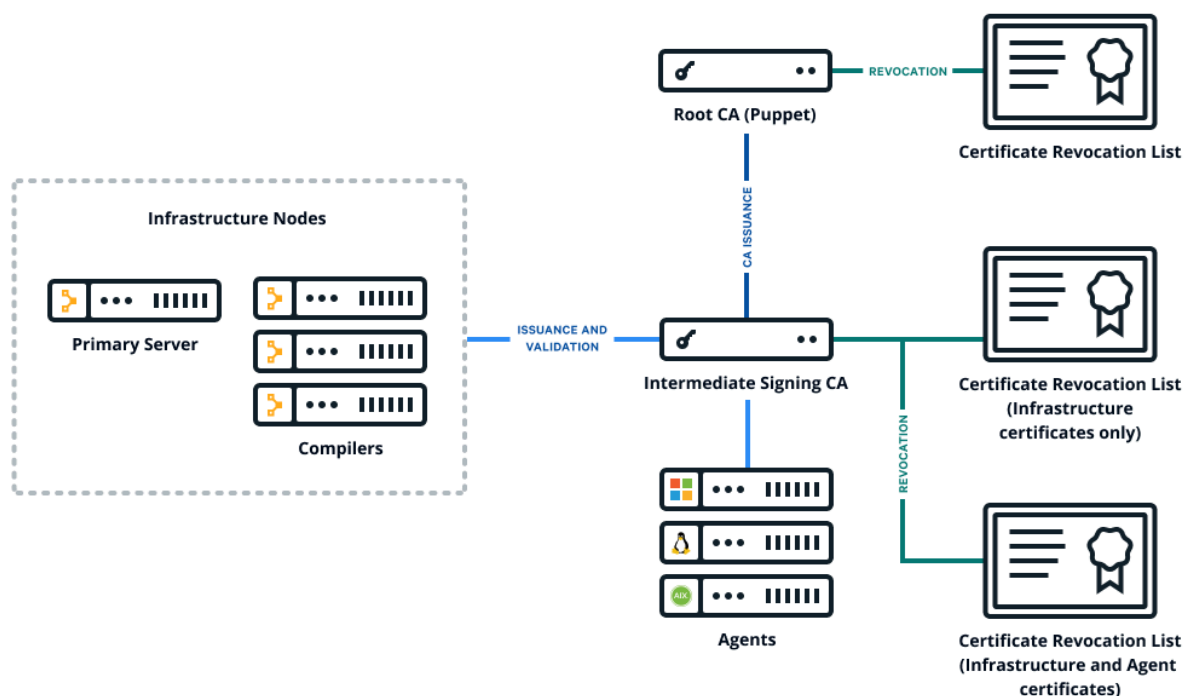
Signing certificates with subject alternative names or auth extensions

With the removal of `puppet cert sign`, Puppet Server's CA API can sign certificates with subject alternative names (SANs) or auth extensions, which was previously disallowed. This option is disabled by default for security reasons, but you can turn it on by setting `allow-subject-alt-names` or `allow-authorization-extensions` to `true` in the `certificate-authority` section of Puppet Server's config (usually located in `ca.conf`). After these have been configured, you can use `puppetserver ca sign --certname <name>` to sign certificates with these additions.

Intermediate CA

Puppet Server supports both a simple CA architecture, with a self-signed root cert that is also used as the CA signing cert; and an intermediate CA architecture, with a self-signed root that issues an intermediate CA cert used for signing incoming certificate requests. The intermediate CA architecture is preferred, because it is more secure and makes regenerating certs easier. To generate a default intermediate CA for Puppet Server, run the `puppetserver ca setup` command before starting your server for the first time.

The following diagram shows the configuration of Puppet's basic certificate infrastructure.



If you have an external certificate authority, you can create a cert chain from it, and use the `puppetserver ca import` subcommand to install the chain on your server. Puppet agents starting with Puppet 6 handle an intermediate CA setup out of the box. No need to copy files around by hand or configure CRL checking. Like `setup`, `import` needs to be run before starting your server for the first time.

Note: The PE installer uses the `puppetserver ca setup` command to create a root cert and an intermediate signing cert for Puppet Server. This means that in PE, the default CA is always an intermediate CA as of PE 2019.0.

Note: If for some reason you cannot use an intermediate CA, in Puppet Server 6 starting the server will generate a non-intermediate CA the same as it always did before the introduction of these commands. However, we don't recommend this, as using an intermediate CA provides more security and easier paths for CA regeneration. It is also the default in PE, and some recommended workflows may rely on it.

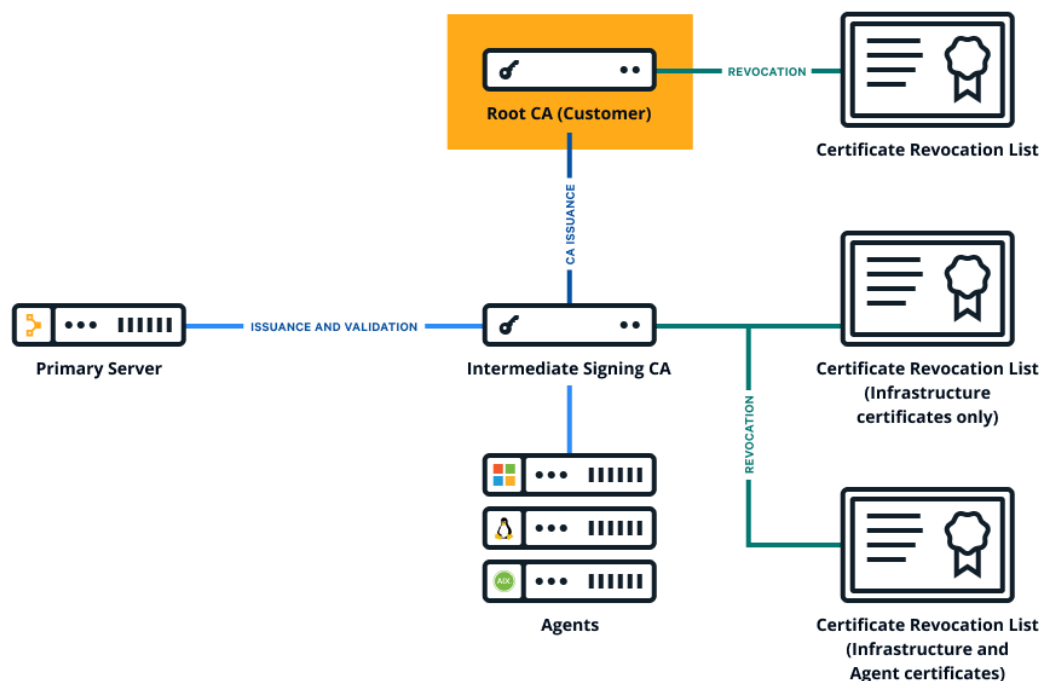
Where to set CA configuration

All CA configuration takes place in Puppet's config file. See the [Puppet Configuration Reference](#) for details.

Set up Puppet as an intermediate CA with an external root

Puppet Server needs to present the full certificate chain to clients so the client can authenticate the server. You construct the certificate chain by concatenating the CA certificates, starting with the new intermediate CA certificate and descending to the root CA certificate.

The following diagram shows the configuration of Puppet's certificate infrastructure with an external root.



To set up Puppet as an intermediate CA with an external root:

1. Collect the PEM-encoded certificates and CRLs for your organization's chain of trust, including the root certificate, any intermediate certificates, and the signing certificate. (The signing certificate might be the root or intermediate certificate.)
2. Create a private RSA key, with no passphrase, for the Puppet CA.
3. Create a PEM-encoded Puppet CA certificate.
 - a. Create a CSR for the Puppet CA.
 - b. Generate the Puppet CA certificate by signing the CSR using your external CA.

Ensure the CA constraint is set to true and the keyIdentifier is composed of the 160-bit SHA-1 hash of the value of the bit string subjectPublicKeyfield. See RFC 5280 section 4.2.1.2 for details.

4. Concatenate all of the certificates into a PEM-encoded certificate bundle, starting with the Puppet CA cert and ending with your root certificate.

```
-----BEGIN CERTIFICATE-----
<Puppet's CA cert>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Org's intermediate CA signing cert>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Org's root CA cert>
-----END CERTIFICATE-----
```

5. Concatenate all of the CRLs into a PEM-encoded CRL chain, starting with any optional intermediate CA CRLs and ending with your root certificate CRL.

```
-----BEGIN X509 CRL-----
<Puppet's CA CRL>
```

```

-----END X509 CRL-----
-----BEGIN X509 CRL-----
<Org's intermediate CA CRL>
-----END X509 CRL-----
-----BEGIN X509 CRL-----
<Org's root CA CRL>
-----END X509 CRL-----

```

6. Use the `puppetserver ca import` command to trigger the rest of the CA setup:

```
puppetserver ca import --cert-bundle ca-bundle.pem --crl-chain crls.pem --
private-key puppet_ca_key.pem
```

7. optional.

```

openssl x509 -in /etc/puppetlabs/puppet/ssl/ca/signed/<HOSTNAME>.crt
-text -noout
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 1 (0x1)
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: CN=intermediate-ca

```

Note: Puppet 5 agents still do not support intermediate CAs. If you must use a Puppet 5 agent with a new (or regenerated) Puppet 6 CA, follow the [instructions](#) for setting up Puppet 5 agents for intermediate CAs.

Autosigning certificate requests

Before Puppet agent nodes can retrieve their configuration catalogs, they require a signed certificate from the local Puppet certificate authority (CA). When using Puppet's built-in CA instead of an external CA, agents submit a certificate signing request (CSR) to the CA to retrieve a signed certificate after it's available.

By default, these CSRs must be manually signed by an admin user, using either the `puppetserver ca` command or the **Node requests** page in the Puppet Enterprise console.

Alternatively, to speed up the process of bringing new agent nodes into the deployment, you can configure the CA to automatically sign certain CSRs.



CAUTION: Autosigning CSRs changes the nature of your deployment's security, and you should understand the implications before configuring it. Each type of autosigning has its own security impact.

Disabling autosigning

By default, the `autosign` setting in the `[server]` section of the CA's `puppet.conf` file is set to `$confdir/autosign.conf`. The basic autosigning functionality is enabled upon installation.

Depending on your installation method, there might not be an allowlist at that location after the Puppet Server is running:

- Open source Puppet: `autosign.conf` doesn't exist by default.
- Monolithic Puppet Enterprise (PE) installations: All required services run on one server, and `autosign.conf` exists on the primary server, but by default it's empty because the primary server doesn't need to add other servers to an allowlist.
- Split PE installations: Services like PuppetDB can run on different servers, the `autosign.conf` exists on the CA server and contains an allowlist of other required hosts.

If the `autosign.conf` file is empty or doesn't exist, the allowlist is effectively empty. The CA Puppet primary server doesn't autosign any certificates until the `autosign` setting's path is configured, or until the default `autosign.conf` file is a non-executable allowlist 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 `[server]` section of the CA Puppet primary server'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 `[server]` section of the CA Puppet primary server'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 an allowlist of certificate names and domain name globs. When a CSR arrives, the requested certificate name is checked against the allowlist 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` allowlist 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 `[server]` 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` allowlist.

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 allowlist is effectively empty.

The CA Puppet primary server therefore doesn't autosign any certificates until the `autosign.conf` file contains a properly formatted allowlist or is a custom policy executable that the Puppet user has permission to run, or until the `autosign` setting is pointed at an allowlist file with properly formatted content or a custom policy executable that the Puppet user has permission to run.

Security implications of basic autosigning

Basic autosigning is insecure because any host can provide any certname when requesting a certificate. Use it only when you fully trust any computer capable of connecting to the Puppet primary server.

With basic autosigning enabled, an attacker who guesses an unused certname allowed by `autosign.conf` can obtain a signed agent certificate from the Puppet primary server. The attacker could then obtain a configuration catalog, which can contain sensitive information depending on your deployment's Puppet code and node classification.

Policy-based autosigning

In policy-based autosigning, the CA runs an external policy executable every time it receives a CSR. This executable examines the CSR and tells the CA whether the certificate is approved for autosigning. If the executable approves, the certificate is autosigned; if not, it's left for manual review.

Enabling policy-based autosigning

To enable policy-based autosigning, set `autosign = <policy executable file>` in the `[server]` section of the CA Puppet primary server's `puppet.conf`.

The policy executable file must be executable by the same user as the Puppet primary server. If not, it is treated as a certname allowlist 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 primary server passes it the certname of the request (as a command line argument) and the PEM-encoded CSR (on stdin), and expects a 0 (approved) or non-zero (rejected) exit code.

After it has the CSR, a policy executable can extract information from it and decide whether to approve the certificate for autosigning. This is useful when you are provisioning your nodes and are [embedding additional information in the CSR](#).

If you aren't embedding additional data, the CSR contains only the node's certname and public key. This can still provide more flexibility and security than `autosign.conf`, as the executable can do things like query your provisioning system, CMDB, or cloud provider to make sure a node with that name was recently added.

Security implications of policy-based autosigning

Depending on how you manage the information the policy executable is using, policy-based autosigning can be fast and extremely secure.

For example:

- If you embed a unique pre-shared key on each node you provision, and provide your policy executable with a database of these keys, your autosigning security is as good as your handling of the keys. As long as it's impractical for an attacker to acquire a PSK, it's impractical for them to acquire a signed certificate.
- If nodes running on a cloud service embed their instance UUIDs in their CSRs, and your executable queries the cloud provider's API to check that a node's UUID exists in your account, your autosigning security is as good as the security of the cloud provider's API. If an attacker can impersonate a legit user to the API and get a list of node UUIDs, or if they can create a rogue node in your account, they can acquire a signed certificate.

When designing your CSR data and signing policy, you must think things through carefully. If you can arrange reasonable end-to-end security for secret data on your nodes, you can configure a secure autosigning system.

Policy executable API

The API for policy executables is as follows.

Run environment	<ul style="list-style-type: none"> • The executable runs one time for each incoming CSR. • It is executed by the Puppet primary server process and runs as the same user as the Puppet primary server. • The Puppet primary server process is blocked until the executable finishes running. We expect policy executables to finish in a timely fashion; if they do not, it's possible for them to tie up all available Puppet primary server threads and deny service to other agents. If an executable needs to perform network requests or other potentially expensive operations, the author is in charge of implementing any necessary timeouts, possibly bailing and exiting non-zero in the event of failure.
-----------------	---

Arguments	<ul style="list-style-type: none"> • The executable must allow a single command line argument. This argument is the Subject CN (certname) of the incoming CSR. • No other command line arguments should be provided. • The Puppet primary server should never fail to provide this argument.
Stdin	<ul style="list-style-type: none"> • The executable receives the entirety of the incoming CSR on its stdin stream. The CSR is encoded in pem format. • The stdin stream contains nothing but the complete CSR. • The Puppet primary server should never fail to provide the CSR on stdin.
Exit status	<ul style="list-style-type: none"> • 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 primary server treats all non-zero exit statuses as equivalent.
Stdout and stderr	<ul style="list-style-type: none"> • Anything the executable emits on stdout or stderr is copied to the Puppet Server log output at the debug log level. Puppet otherwise ignores the executable's output; only the exit code is considered significant.

CSR attributes and certificate extensions

When Puppet agent nodes request their certificates, the certificate signing request (CSR) usually contains only their certname and the necessary cryptographic information. Agents can also embed additional data in their CSR, useful for policy-based autosigning and for adding new trusted facts.

Embedding additional data into CSRs is useful when:

- Large numbers of nodes are regularly created and destroyed as part of an elastic scaling system.
- You are willing to build custom tooling to make certificate autosigning more secure and useful.

It might also be useful in deployments where Puppet is used to deploy private keys or other sensitive information, and you want extra control over nodes that receive this data.

If your deployment doesn't match one of these descriptions, you might not need this feature.

Timing: When data can be added to CSRs and certificates

When Puppet agent starts the process of requesting a catalog, it checks whether it has a valid signed certificate. If it does not, it generates a key pair, crafts a CSR, and submits it to the certificate authority (CA) Puppet Server. For detailed information, see [agent/server HTTPS traffic](#).

For practical purposes, a certificate is locked and immutable as soon as it is signed. For data to persist in the certificate, it has to be added to the CSR before the CA signs the certificate.

This means any desired extra data must be present before Puppet agent attempts to request its catalog for the first time.

Populate any extra data when provisioning the node. If you make an error, see the Troubleshooting section below for information about recovering from failed data embedding.

Data location and format

Extra data for the CSR is read from the `csr_attributes.yaml` file in Puppet's `confdir`. The location of this file can be changed with the `csr_attributes` configuration setting.

The `csr_attributes.yaml` file must contain a YAML hash with one or both of the following keys:

- `custom_attributes`
- `extension_requests`

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

- Each key is a valid [object identifier \(OID\)](#) — [Puppet-specific OIDs](#) can optionally be referenced by short name instead of by numeric ID.
- Each value is an object that can be cast to a string — numbers are allowed but arrays are not.

For information about how each hash is used and recommended OIDs for each hash, see the sections below.

Custom attributes (transient CSR data)

Custom attributes are pieces of data that are embedded only in the CSR. The CA can use them when deciding whether to sign the certificate, but they are discarded after that and aren't transferred to the final certificate.

Default behavior

The `puppetserver ca list` command doesn't display custom attributes for pending CSRs, and [basic autosigning \(autosign.conf\)](#) doesn't check them before signing.

Configurable behavior

If you use [policy-based autosigning](#) your policy executable receives the complete CSR in PEM format. The executable can extract and inspect the custom attributes, and use them to decide whether to sign the certificate.

The simplest method is to embed a pre-shared key of some kind in the custom attributes. A policy executable can compare it to a list of known keys and autosign certificates for any pre-authorized nodes.

A more complex use might be to embed an instance-specific ID and write a policy executable that can check it against a list of your recently requested instances on a public cloud, like EC2 or GCE.

Manually checking for custom attributes in CSRs

You can check for custom attributes by using OpenSSL to dump a CSR in pem format to text format, by running this command:

```
openssl req -noout -text -in <name>.pem
```

In the output, look for the `Attributes` section which appears below the `Subject Public Key Info` block:

```
Attributes:
  challengePassword      : 342thbjkt82094y0uthhor289jnqthpc2290
```

Recommended OIDs for attributes

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

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

You can also use the [Puppet-specific OIDs](#).

Extension requests (permanent certificate data)

Extension requests are pieces of data that are transferred as extensions to the final certificate, when the CA signs the CSR. They persist as trusted, immutable data, that cannot be altered after the certificate is signed.

They can also be used by the CA when deciding whether or not to sign the certificate.

Default behavior

When signing a certificate, Puppet's CA tools transfer any extension requests into the final certificate.

You can access certificate extensions in manifests as `$trusted["extensions"] ["<EXTENSION OID>"]`.

Select OIDs in the `ppRegCertExt` and `ppAuthCertExt` ranges. See the [Puppet-specific Registered IDs](#). By default, any other OIDs appear as plain dotted numbers, but you can use the `custom_trusted_oid_mapping.yaml` file to assign short names to any other OIDs you use at your site. If you do, those OIDs appear in `$trusted` as their short names, instead of their full numerical OID.

For more information about `$trusted`, see [Facts and built-in variables](#).

The visibility of extensions is limited:

- The `puppetserver ca list` command does not display custom attributes for any pending CSRs, and [basic autosigning](#) (`autosign.conf`) doesn't check them before signing. Either use [policy-based autosigning](#) or inspect CSRs manually with the `openssl` command (see below).

Puppet's authorization system (`auth.conf`) does not use certificate extensions, but [Puppet Server's authorization system](#), which is based on `trapperkeeper-authorization`, can use extensions in the `ppAuthCertExt` OID range, and requires them for requests to write access rules.

Configurable behavior

If you use [policy-based autosigning](#), your policy executable receives the complete CSR in pem format. The executable can extract and inspect the extension requests, and use them when deciding whether to sign the certificate.

Manually checking for extensions in CSRs and certificates

You can check for extension requests in a CSR by running the OpenSSL command to dump a CSR in pem format to text format:

```
openssl req -noout -text -in <name>.pem
```

In the output, look for a section called `Requested Extensions`, which appears below the `Subject Public Key Info` and `Attributes` blocks:

```
Requested Extensions:
  pp_uuid:
    . $ED803750-E3C7-44F5-BB08-41A04433FE2E
  1.3.6.1.4.1.34380.1.1.3:
    ..my_ami_image
  1.3.6.1.4.1.34380.1.1.4:
    . $342thbjkt82094y0uthhor289jnqthpc2290
```

Note: Every extension is preceded by any combination of two characters (. \$ and . . in the example above) that contain ASN.1 encoding information. Because OpenSSL is unaware of Puppet’s custom extensions OIDs, it’s unable to properly display the values.

Any Puppet-specific OIDs (see below) appear as numeric strings when using OpenSSL.

You can check for extensions in a signed certificate by running:

```
/opt/puppetlabs/puppet/bin/openssl x509 -noout -text -in $(puppet config
print signedir)/<certname>.pem
```

In the output, look for the X509v3 extensions section. Any of the Puppet-specific [registered OIDs](#) appear as their descriptive names:

```
X509v3 extensions:
  Netscape Comment:
    Puppet Ruby/OpenSSL Internal Certificate
  X509v3 Subject Key Identifier:
    47:BC:D5:14:33:F2:ED:85:B9:52:FD:A2:EA:E4:CC:00:7F:7F:19:7E
  Puppet Node UUID:
    ED803750-E3C7-44F5-BB08-41A04433FE2E
  X509v3 Extended Key Usage: critical
    TLS Web Server Authentication, TLS Web Client Authentication
  X509v3 Basic Constraints: critical
    CA:FALSE
  Puppet Node Preshared Key:
    342thbjkt82094y0uthhor289jnqthpc2290
  X509v3 Key Usage: critical
    Digital Signature, Key Encipherment
  Puppet Node Image Name:
    my_ami_image
```

Recommended OIDs for extensions

Extension request OIDs must be under the ppRegCertExt (1.3.6.1.4.1.34380.1.1), ppPrivCertExt (1.3.6.1.4.1.34380.1.2), or ppAuthCertExt (1.3.6.1.4.1.34380.1.3) OID arcs.

Puppet provides several registered OIDs (under ppRegCertExt) for the most common kinds of extension information, a private OID range (ppPrivCertExt) for site-specific extension information, and an OID range for safe authorization to Puppet Server (ppAuthCertExt).

There are several benefits to using the registered OIDs:

- You can reference them in the `csr_attributes.yaml` file with their short names instead of their numeric IDs.
- You can access them in `$trusted[extensions]` with their short names instead of their numeric IDs.
- When using Puppet tools to print certificate info, they appear using their descriptive names instead of their numeric IDs.

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

You can use the [custom_trusted_oid_mapping.yaml](#) file to set short names for any private extension OIDs you use. Note that this enables only the short names in the `$trusted[extensions]` hash.

Puppet-specific registered IDs

ppRegCertExt

The ppRegCertExt OID range contains the following OIDs:

Numeric ID	Short name	Descriptive name
1.3.6.1.4.1.34380.1.1.1	pp_uuid	Puppet node UUID
1.3.6.1.4.1.34380.1.1.2	pp_instance_id	Puppet node instance ID
1.3.6.1.4.1.34380.1.1.3	pp_image_name	Puppet node image name
1.3.6.1.4.1.34380.1.1.4	pp_preshared_key	Puppet node preshared key
1.3.6.1.4.1.34380.1.1.5	pp_cost_center	Puppet node cost center name
1.3.6.1.4.1.34380.1.1.6	pp_product	Puppet node product name
1.3.6.1.4.1.34380.1.1.7	pp_project	Puppet node project name
1.3.6.1.4.1.34380.1.1.8	pp_application	Puppet node application name
1.3.6.1.4.1.34380.1.1.9	pp_service	Puppet node service name
1.3.6.1.4.1.34380.1.1.10	pp_employee	Puppet node employee name
1.3.6.1.4.1.34380.1.1.11	pp_created_by	Puppet node created_by tag
1.3.6.1.4.1.34380.1.1.12	pp_environment	Puppet node environment name
1.3.6.1.4.1.34380.1.1.13	pp_role	Puppet node role name
1.3.6.1.4.1.34380.1.1.14	pp_software_version	Puppet node software version
1.3.6.1.4.1.34380.1.1.15	pp_department	Puppet node department name
1.3.6.1.4.1.34380.1.1.16	pp_cluster	Puppet node cluster name
1.3.6.1.4.1.34380.1.1.17	pp_provisioner	Puppet node provisioner name
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. For PE internal use only.

Cloud provider attributes and extensions population example

To populate the `csr_attributes.yaml` file when you provision a node, use an automated script such as `cloud-init`.

For example, when provisioning a new node from the AWS EC2 dashboard, enter the following script into the **Configure Instance Details** —> **Advanced Details** section:

```
#!/bin/sh
if [ ! -d /etc/puppetlabs/puppet ]; then
  mkdir /etc/puppetlabs/puppet
fi
cat > /etc/puppetlabs/puppet/csr_attributes.yaml << YAML
custom_attributes:
  1.2.840.113549.1.9.7: mySuperAwesomePassword
extension_requests:
  pp_instance_id: $(curl -s http://169.254.169.254/latest/meta-data/instance-id)
  pp_image_name: $(curl -s http://169.254.169.254/latest/meta-data/ami-id)
YAML
```

This populates the attributes file with the AWS instance ID, image name, and a pre-shared key to use with policy-based autosigning.

Troubleshooting

Recovering from failed data embedding

When testing this feature for the first time, you might not embed the right information in a CSR, or certificate, and might want to start over for your test nodes. This is not really a problem after your provisioning system is changed to populate the data, but it can easily happen when doing things manually.

To start over, do the following.

On the test node:

- Turn off Puppet agent, if it's running.
- If using Puppet version 6.0.3 or greater, run `puppet ssl clean`. If not, delete the following files:
 - `$ssldir/certificate_requests/<name>.pem`
 - `$ssldir/certs/<name>.pem`

On the CA primary Puppet server:

- Check whether a signed certificate exists. Use `puppetserver ca list --all` to see the complete list. If it exists, revoke and delete it with `puppetserver ca clean --certname <name>`.

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

Regenerating certificates in a Puppet deployment

In some cases, you might need to regenerate the certificates and security credentials (private and public keys) that are generated by Puppet's built-in PKI systems.

For example, you might have a Puppet primary server you need to move to a different network in your infrastructure, or you might have experienced a security vulnerability that makes existing credentials untrustworthy.

Note: There are other, more automated ways of doing this. We recommend using Bolt to regenerate certs when needed. See the [Bolt documentation](#) for more information. There is also a [puppetlabs-certregen](#) module but is not supported with Puppet Server 6.

Important: The information on this page describes the steps for regenerating certs in an open source Puppet deployment. If you use Puppet Enterprise do not use the information on this page, as it leaves you with an incomplete replacement and non-functional deployment. Instead, PE customers must refer to one of the following pages:

- [Regenerating certificates in split PE deployments](#)
- [Regenerating certificates in monolithic PE deployments](#)

If your goal is to...	Do this...
Regenerate an agent's certificate	Clear and regenerate certs for Puppet agents
Fix a compromised or damaged certificate authority	Regenerate the CA and all certificates
Completely regenerate all Puppet deployment certificates	Regenerate the CA and all certificates
Add DNS alt-names or other certificate extensions to your existing Puppet primary server	Regenerate the agent certificate of your Puppet primary server and add DNS alt-names or other certificates

Regenerate the agent certificate of your Puppet primary server and add DNS alt-names or other certificate extensions

This option preserves the primary server/agent relationship and lets you add DNS alt-names or certificate extensions to your existing primary server.

1. Revoke the Puppet primary server's certificate and clean the CA files pertaining to it. Note that the agents won't be able to connect to the primary server until all of the following steps are finished.

```
puppetserver ca clean --certname <CERTNAME_OF_YOUR_SERVER>
```

2. Remove the agent-specific copy of the public key, private key, and certificate-signing request pertaining to the certificate:

```
puppet ssl clean
```

3. Stop the Puppet primary server service:

```
puppet resource service puppetserver ensure=stopped
```

Note: The CA and server run in the same primary server so this also stops the CA.

4. After you've stopped the primary server and CA service, create a certificate signed by the CA and add DNS alt names (comma separated):

```
puppetserver ca generate --certname <CERTNAME> --subject-alt-names <DNS ALT NAMES> --ca-client
```

Note: If you don't want to add DNS alt names to your primary server, omit the `--subject-alt-names <DNS ALT NAMES>` option from the command above.

5. Restart the Puppet primary server service:

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

Regenerate the CA and all certificates



CAUTION: This process destroys the certificate authority and all other certificates. It is meant for use in the event of a total compromise of your site, or some other unusual circumstance. If you want to preserve the primary server/agent relationship, [regenerate the agent certificate of your Puppet primary server](#). If you just need to replace a few agent certificates, [clear and regenerate certs for Puppet agents](#).

Step 1: Clear and regenerate certs on your primary Puppet server

On the primary server 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 primary server 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 primary server's cert:

```
sudo puppetserver ca setup
```

You will see this message: Notice: Signed certificate request for ca.

6. Start the primary server service by running:

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

7. Start the Puppet agent service by running this command:

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

At this point:

- You have a new CA certificate and key.
- Your primary server has a certificate from the new CA, and it can field new certificate requests.
- The primary server 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 primary server 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 primary Puppet server. For each extension like this, you'll need to regenerate the certificates it uses.

Many tools have scripts or documentation to help you set up SSL, and you can often just re-run the setup instructions.

PuppetDB

We recommend PuppetDB users first follow the instructions in Step 3: Clear and regenerate certs for agents, below, because PuppetDB re-uses Puppet agents' certificates. After that, restart the PuppetDB service. See [Redo SSL setup after changing certificates](#) for more information.

Step 3: Clear and regenerate certs for Puppet agents

To replace the certs on agents, you'll need to log into each agent node and do the following steps.

1. Stop the agent service. On *nix:

```
sudo puppet resource service puppet ensure=stopped
```

On Windows, with Administrator privileges:

```
puppet resource service puppet ensure=stopped
```

2. Locate Puppet's [SSL directory](#) and delete its contents.

The SSL directory can be determined by running `puppet config print ssldir --section agent`

3. Restart the agent service. On *nix:

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

On Windows, with Administrator privileges:

```
puppet resource service puppet ensure=running
```

When the agent starts, it generates keys and requests a new certificate from the CA primary server.

4. If you are not using autosigning, log in to the CA primary server and sign each agent node's certificate request.

To view pending requests, run:

```
sudo puppetserver ca list
```

To sign requests, run:

```
sudo puppetserver ca sign --certname <NAME>
```

After an agent node's new certificate is signed, it's retrieved within a few minutes and a Puppet run starts.

After you have regenerated all agents' certificates, everything will be fully functional under the new CA.

Note: You can achieve the same results by turning these steps into Bolt tasks or plans. See the [Bolt documentation](#) for more information.

External CA

This information describes the supported and tested configurations for external CAs in this version of Puppet. If you have an external CA use case that isn't listed here, contact Puppet so we can learn more about it.

Supported external CA configurations

This version of Puppet supports some external CA configurations, however not every possible configuration is supported.

We fully support the following setup options:

- Single CA which directly issues SSL certificates.
- Puppet Server functioning as an intermediate CA.

Fully supported by Puppet means:

- If issues arise that are considered bugs, we'll fix them as soon as possible.
- If issues arise in any other external CA setup that are considered feature requests, we'll consider whether to expand our support.

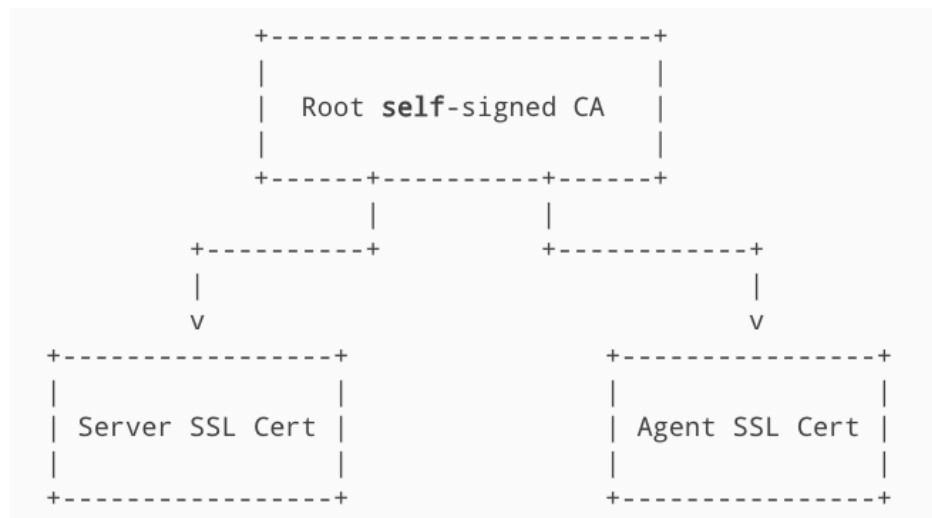
Option 1: Puppet Server functioning as an intermediate CA

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

See [Using Puppet Server as an intermediate certificate authority](#).

Option 2: Single CA

When Puppet uses its internal CA, it defaults to a single CA configuration. A single externally issued CA can also be used in a similar manner.



This is an all or nothing configuration rather than a mix-and-match. When using an external CA, the built-in Puppet CA service must be disabled and cannot be used to issue SSL certificates.

Note: Puppet cannot automatically distribute certificates in this configuration.

Puppet Server

Configure Puppet Server in three steps:

- Disable the internal CA service.
 - Ensure that the certname does not change.
 - Put certificates and keys in place on disk.
1. To disable the internal CA, edit the Puppet Server/etc/puppetlabs/puppetserver/services.d/ca.cfg file to comment and uncomment the appropriate settings:

```
# To enable the CA service, leave the following line uncommented
puppetlabs.services.ca.certificate-authority-service/certificate-
authority-service
# To disable the CA service, comment out the above line and uncomment the
line below
#puppetlabs.services.ca.certificate-authority-disabled-service/
certificate-authority-disabled-service
puppetlabs.trapperkeeper.services watcher.filesystem-watch-service/
filesystem-watch-service
```

2. Set a static value for the certname setting in puppet.conf:

```
[server]
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.

- 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	<code>puppet config print hostcert --section server</code>
Server SSL certificate private key	<code>puppet config print hostprivkey --section server</code>
Root CA certificate	<code>puppet config print localcacert --section server</code>
Root certificate revocation list	<code>puppet config print hostcrl --section server</code>

If you've put the credentials in the correct locations, you don't need to change any additional settings.

Puppet agent

You don't need to change any settings. Put the external credentials into the correct filesystem locations. You can run the following commands to find the appropriate locations.

Credential	File location
Agent SSL certificate	<code>puppet config print hostcert --section agent</code>
Agent SSL certificate private key	<code>puppet config print hostprivkey --section agent</code>
Root CA certificate	<code>puppet config print localcacert --section agent</code>
Root certificate revocation list	<code>puppet config print hostcrl --section agent</code>

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 server --configprint hostcert`. Equivalent to the 'SSLCertificateFile' Apache config setting.
- `ssl-key`: The value of `puppet server --configprint hostprivkey`. Equivalent to the 'SSLCertificateKeyFile' Apache config setting.
- `ssl-ca-cert`: The value of `puppet server --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/server.pem
  ssl-key     : /path/to/server.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, see [Configuring 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.

External SSL termination

Use the following steps to configure external SSL termination.

Disable HTTPS for Puppet Server

You'll need to turn off SSL and have Puppet Server use the HTTP protocol instead: remove the `ssl-port` and `ssl-host` settings from the `conf.d/webserver.conf` file and replace them with `port` and `host` settings. See [Configuring the Webserver Service](#) for more information on configuring the web server service.

Allow Client Cert Data From HTTP Headers

When using external SSL termination, Puppet Server expects to receive client certificate information via some HTTP headers.

By default, reading this data from headers is disabled. To allow Puppet Server to recognize it, you'll need to set `allow-header-cert-info: true` in the authorization config section of the `/etc/puppetlabs/puppetserver/conf.d/auth.conf` file.

See [Configuring Puppet Server](#) on page 72 for more information on the `puppetserver.conf` and `auth.conf` files.

Note: This assumes the default behavior of Puppet 5 and greater of using Puppet Server's `hocon` `auth.conf` rather than Puppet's older `ini`-style `auth.conf`.

WARNING: Setting `allow-header-cert-info` to 'true' puts Puppet Server in an incredibly vulnerable state. Take extra caution to ensure it is **absolutely not reachable** by an untrusted network.

With `allow-header-cert-info` set to 'true', authorization code will use only the client HTTP header values---not an SSL-layer client certificate---to determine the client subject name, authentication status, and trusted facts. This is true even if the web server is hosting an HTTPS connection. This applies to validation of the client via rules in the `auth.conf` file and any [trusted facts](#) extracted from certificate extensions.

If the `client-auth` setting in the `webserver` config block is set to `need` or `want`, the Jetty web server will still validate the client certificate against a certificate authority store, but it will only verify the SSL-layer client certificate---not a certificate in an `X-Client-Cert` header.

Reload Puppet Server

You'll need to reload Puppet Server for the configuration changes to take effect.

Configure SSL Terminating Proxy to Set HTTP Headers

The device that terminates SSL for Puppet Server must extract information from the client's certificate and insert that information into three HTTP headers. See the documentation for your SSL terminator for details.

The headers you'll need to set are `X-Client-Verify`, `X-Client-DN`, and `X-Client-Cert`.

X-Client-Verify

Mandatory. Must be either `SUCCESS` if the certificate was validated, or something else if not. (The convention seems to be to use `NONE` for when a certificate wasn't presented, and `FAILED:reason` for other validation failures.) Puppet Server uses this to authorize requests; only requests with a value of `SUCCESS` will be considered authenticated.

X-Client-DN

Mandatory. Must be the [Subject DN](#) of the agent's certificate, if a certificate was presented. Puppet Server uses this to authorize requests.

X-Client-Cert

Optional. Should contain the client's [PEM-formatted](#) (Base-64) certificate (if a certificate was presented) in a single URI-encoded string. Note that URL encoding is not sufficient; all space characters must be encoded as `%20` and not `+` characters.

Note: Puppet Server only uses the value of this header to extract [trusted facts](#) from extensions in the client certificate. If you aren't using trusted facts, you can choose to reduce the size of the request payload by omitting the `X-Client-Cert` header.

Note: Apache's `mod_proxy` converts line breaks in PEM documents to spaces for some reason, and Puppet Server can't decode the result. We're tracking this issue as [SERVER-217](#).

PuppetDB

All of the data generated by Puppet (for example facts, catalogs, reports) is stored in PuppetDB.

Storing data in PuppetDB allows Puppet to work faster and provides an API for other applications to access Puppet's collected data. Once PuppetDB is full of your data, it becomes a great tool for infrastructure discovery, compliance reporting, vulnerability assessment, and more. You perform all of these tasks with PuppetDB queries.

See the [PuppetDB docs](#) for more information.

Facter

Facter is Puppet's cross-platform system profiling library. It discovers and reports per-node facts, which are available in your Puppet manifests as variables.

Facter is published as a gem to <https://rubygems.org/>. If you've already got Ruby installed, you can install Facter by running:

```
gem install facter
```

- [Factor release notes](#) on page 29

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

- [Factor CLI](#)
- [Core facts](#)
- [Custom facts overview](#) on page 224

You can add custom facts by writing snippets of Ruby code on the primary Puppet server. Puppet then uses plug-ins in modules to distribute the facts to the client.

- [Writing custom facts](#) on page 230

A typical fact in Factor is a collection of several elements, and is written either as a simple value (“flat” fact) or as structured data (“structured” fact). This page shows you how to write and format facts correctly.

- [External facts](#) on page 235

External facts provide a way to use arbitrary executables or scripts as facts, or set facts statically with structured data. With this information, you can write a custom fact in Perl, C, or a one-line text file.

- [Configuring Factor with `factor.conf`](#) on page 238

The `factor.conf` file is a configuration file that allows you to cache and block fact groups and facts, and manage how Factor interacts with your system. There are four sections: `facts`, `global`, `cli` and `fact-groups`. All sections are optional and can be listed in any order within the file.

Custom facts overview

You can add custom facts by writing snippets of Ruby code on the primary Puppet server. Puppet then uses plug-ins in modules to distribute the facts to the client.

For information on how to add custom facts to modules, see [Module plug-in types](#).

Related information

[External facts](#) on page 235

External facts provide a way to use arbitrary executables or scripts as facts, or set facts statically with structured data. With this information, you can write a custom fact in Perl, C, or a one-line text file.

[Plug-ins in modules](#) on page 596

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.

Adding custom facts to Factor

Sometimes you need to be able to write conditional expressions based on site-specific data that just isn’t available via Factor, or perhaps you’d like to include it in a template.

Because you can’t include arbitrary Ruby code in your manifests, the best solution is to add a new fact to Factor. These additional facts can then be distributed to Puppet clients and are available for use in manifests and templates, just like any other fact is.

Note: Factor 4 implements the same [custom facts API](#) as Factor 3. Any custom fact that requires one of the Ruby files previously stored in `lib/factor/util` fails with an error.

Structured and flat facts

A typical fact extracts a piece of information about a system and returns it as either as a simple value (“flat” fact) or data organized as a hash or array (“structured” fact). There are several types of facts classified by how they collect information, including:

- Core facts, which are built into Factor and are common to almost all systems.
- Custom facts, which run Ruby code to produce a value.
- External facts, which return values from pre-defined static data, or the result of an executable script or program.

All fact types can produce flat or structured values.

Related information

[Factor release notes](#) on page 29

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

[External facts](#) on page 235

External facts provide a way to use arbitrary executables or scripts as facts, or set facts statically with structured data. With this information, you can write a custom fact in Perl, C, or a one-line text file.

Loading custom facts

Facter offers multiple methods of loading facts.

These include:

- `$LOAD_PATH`, or the Ruby library load path.
- The `--custom-dir` command line option.
- The environment variable `FACTERLIB`.

You can use these methods to do things like test files locally before distributing them, or you can arrange to have a specific set of facts available on certain machines.

Using the Ruby load path

Facter searches all directories in the Ruby `$LOAD_PATH` variable for subdirectories named Facter, and loads all Ruby files in those directories. If you had a directory in your `$LOAD_PATH` like `~/lib/ruby`, set up like this:

```
#~/lib/ruby
### facter
### rackspace.rb
### system_load.rb
### users.rb
```

Facter loads `facter/system_load.rb`, `facter/users.rb`, and `facter/rackspace.rb`.

Using the `--custom-dir` command line option

Facter can take multiple `--custom-dir` options on the command line that specifies a single directory to search for custom facts. Facter attempts to load all Ruby files in the specified directories. This allows you to do something like this:

```
$ ls my_facts
system_load.rb
$ ls my_other_facts
users.rb
$ facter --custom-dir=./my_facts --custom-dir=./my_other_facts system_load
users
system_load => 0.25
users => thomas,pat
```

Using the `FACTERLIB` environment variable

Facter also checks the environment variable `FACTERLIB` for a delimited (semicolon for Windows and colon for all other platforms) set of directories, and tries to load all Ruby files in those directories. This allows you to do something like this:

```
$ ls my_facts
system_load.rb
$ ls my_other_facts
users.rb
$ export FACTERLIB="./my_facts:./my_other_facts"
$ facter system_load users
system_load => 0.25
users => thomas,pat
```

Note: Facter 4 replaces `facter -p` with `puppet facts show`. The `puppet facts show` command is the default action for Puppet facts. Facter also accepts `puppet facts`.

Two parts of every fact

Most facts have at least two elements.

1. A call to `Facter.add('fact_name')`, which determines the name of the fact.
2. A `setcode` statement for simple resolutions, which is evaluated to determine the fact's value.

Facts can get a lot more complicated than that, but those two together are the most common implementation of a custom fact.

Executing shell commands in facts

Puppet gets information about a system from Facter, and the most common way for Facter to get that information is by executing shell commands.

You can then parse and manipulate the output from those commands using standard Ruby code. The Facter API gives you a few ways to execute shell commands:

- To run a command and use the output verbatim, as your fact's value, you can pass the command into `setcode` directly. For example: `setcode 'uname --hardware-platform'`
- If your fact is more complicated than that, you can call `Facter::Core::Execution.execute('uname --hardware-platform')` from within the `setcode do ... end` block. Whatever the `setcode` statement returns is used as the fact's value.
- Your shell command is also a Ruby string, so you need to escape special characters if you want to pass them through.

Note: Not everything that works in the terminal works in a fact. You can use the pipe (`|`) and similar operators as you normally would, but Bash-specific syntax like `if` statements do not work. The best way to handle this limitation is to write your conditional logic in Ruby.

Example

To get the output of `uname --hardware-platform` to single out a specific type of workstation, you create a custom fact.

1. Start by giving the fact a name, in this case, `hardware_platform`.
2. Create the fact in a file called `hardware_platform.rb` on the primary Puppet server:

```
# hardware_platform.rb

Facter.add('hardware_platform') do
  setcode do
    Facter::Core::Execution.execute('/bin/uname --hardware-platform')
  end
end
```

3. Use the instructions in the [Plug-ins in modules docs](#) to copy the new fact to a module and distribute it. During your next Puppet run, the value of the new fact is available to use in your manifests and templates.

Using other facts

You can write a fact that uses other facts by accessing `Facter.value(:somefact)`. If the fact fails to resolve or is not present, Facter returns `nil`.

For example:

```
Facter.add(:osfamily) do
  setcode do
    distid = Facter.value(:lsbdistid)
    case distid
    when /RedHatEnterprise|CentOS|Fedora/
      'redhat'
```

```

    when 'ubuntu'
      'debian'
    else
      distid
    end
  end
end
end

```

Configuring facts

Facts have properties that you can use to customize how they are evaluated.

Confining facts

One of the more commonly used properties is the `confine` statement, which restricts the fact to run only on systems that match another given fact.

For example:

```

Facter.add(:powerstates) do
  confine kernel: 'Linux'
  setcode do
    Facter::Core::Execution.execute('cat /sys/power/states')
  end
end

```

This fact uses `sysfs` on Linux to get a list of the power states that are available on the given system. Because this is available only on Linux systems, we use the `confine` statement to ensure that this fact isn't needlessly run on systems that don't support this type of enumeration.

To confine structured facts like `['os'] ['family']`, you can use `Facter.value`: You can also use a Ruby block:

```

confine :os do |os|
  os['family'] == 'RedHat'
end

```

Fact precedence

A single fact can have multiple *resolutions*, each of which is a different way of determining the value of the fact. It's common to have different resolutions for different operating systems, for example. To add a new resolution to a fact, you add the fact again with a different `setcode` statement.

When a fact has more than one resolution, the first resolution that returns a value other than `nil` sets the fact's value. The way that Facter decides the issue of resolution precedence is the `weight` property. After Facter rules out any resolutions that are excluded because of `confine` statements, the resolution with the highest weight is evaluated first. If that resolution returns `nil`, Facter moves on to the next resolution (by descending weight) until it gets a value for the fact.

By default, the weight of a resolution is the number of `confine` statements it has, so that more specific resolutions take priority over less specific resolutions. External facts have a weight of 1000 — to override them, set a weight above 1000.

```

# Check to see if this server has been marked as a postgres server
Facter.add(:role) do
  has_weight 100
  setcode do
    if File.exist? '/etc/postgres_server'
      'postgres_server'
    end
  end
end
end

```

```
# Guess if this is a server by the presence of the pg_create binary
Facter.add(:role) do
  has_weight 50
  setcode do
    if File.exist? '/usr/sbin/pg_create'
      'postgres_server'
    end
  end
end

# If this server doesn't look like a server, it must be a desktop
Facter.add(:role) do
  setcode do
    'desktop'
  end
end
```

Execution timeouts

Facter 4 supports timeouts on resolutions. If the timeout is exceeded, Facter prints an error message.

```
Facter.add(:foo, {timeout: 0.2}) do
  setcode do
    Facter::Core::Execution.execute("sleep 1")
  end
end
```

You can also pass a timeout to `Facter::Core::Execution#execute`:

```
Facter.add(:sleep) do
  setcode do
    begin
      Facter::Core::Execution.execute('sleep 10', options = {:timeout => 5})
      'did not timeout!'
    rescue Facter::Core::Execution::ExecutionFailure
      Facter.warn("Sleep fact timed out!")
    end
  end
end
```

When Facter runs as standalone, using `Facter.warn` ensures that the message is printed to `STDERR`. When Facter is called as part of a catalog application, using `Facter.warn` prints the message to Puppet's log. If an exception is not caught, Facter automatically logs it as an error.

Structured facts

Structured facts take the form of either a hash or an array.

To create a structured fact, return a hash or an array from the `setcode` statement.

You can see some relevant examples in the [Writing structured facts](#) section of the Custom facts overview.

Facter 4 introduced a new way to aggregate facts using dot notation. For more information, see [Writing aggregate resolutions](#).

Related information

[Writing custom facts](#) on page 230

A typical fact in Factor is an collection of several elements, and is written either as a simple value (“flat” fact) or as structured data (“structured” fact). This page shows you how to write and format facts correctly.

Aggregate resolutions

If your fact combines the output of multiple commands, use aggregate resolutions. An aggregate resolution is split into chunks, each one responsible for resolving one piece of the fact. After all of the chunks have been resolved separately, they’re combined into a single flat or structured fact and returned.

Aggregate resolutions have several key differences compared to simple resolutions, beginning with the fact declaration. To introduce an aggregate resolution, add the `:type => :aggregate` parameter:

```
Factor.add(:fact_name, :type => :aggregate) do
  #chunks go here
  #aggregate block goes here
end
```

Each step in the resolution then gets its own named chunk statement:

```
chunk(:one) do
  'Chunk one returns this. '
end

chunk(:two) do
  'Chunk two returns this.'
end
```

Aggregate resolutions never have a `setcode` statement. Instead, they have an optional `aggregate` block that combines the chunks. Whatever value the aggregate block returns is the fact’s value. Here’s an example that just combines the strings from the two chunks above:

```
aggregate do |chunks|
  result = ''

  chunks.each_value do |str|
    result += str
  end

  # Result: "Chunk one returns this. Chunk two returns this."
  result
end
```

If the chunk blocks all return arrays or hashes, you can omit the `aggregate` block. If you do, Factor merges all of your data into one array or hash and uses that as the fact’s value.

For more examples of aggregate resolutions, see the Aggregate resolutions section of the Custom facts overview page.

Related information

[Writing custom facts](#) on page 230

A typical fact in Factor is an collection of several elements, and is written either as a simple value (“flat” fact) or as structured data (“structured” fact). This page shows you how to write and format facts correctly.

Viewing fact values

If your Puppet primary servers are configured to use PuppetDB, you can view and search all of the facts for any node, including custom facts.

See [the PuppetDB docs](#) for more info.

Writing custom facts

A typical fact in `Facter` is an collection of several elements, and is written either as a simple value (“flat” fact) or as structured data (“structured” fact). This page shows you how to write and format facts correctly.

Note: It’s important to distinguish between **facts** and **resolutions**. A fact is a piece of information about a given node, while a resolution is a way of obtaining that information from the system. That means every fact needs to have **at least one** resolution, and facts that can run on different operating systems may need to have different resolutions for each one. Facts and resolutions are conceptually different, but have similarities. Declaring a second (or more) resolution for a fact looks just like declaring a completely new fact, only with the same name as an existing fact.

You need some familiarity with Ruby to understand most of these examples. For an introduction, see out the [Custom facts overview](#). For information on how to add custom facts to modules, see [Module plug-in types](#).

Writing facts with simple resolutions

Most facts are resolved all at the same time, without any need to merge data from different sources. In that case, the resolution is simple. Both flat and structured facts can have simple resolutions.

Main components of simple resolutions

Simple facts are typically made up of the following parts:

1. A call to `Facter.add(:fact_name):`
 - This introduces a new fact *or* a new resolution for an existing fact with the same name.
 - The name can be either a symbol or a string.
 - The rest of the fact is wrapped in the `add` call’s `do ... end` block.
2. Zero or more `confine` statements:
 - Determine whether the resolution is suitable (and therefore is evaluated).
 - Can either match against the value of another fact or evaluate a Ruby block.
 - If given a symbol or string representing a fact name, a block is required and the block receives the fact’s value as an argument.
 - If given a hash, the keys are expected to be fact names. The values of the hash are either the expected fact values or an array of values to compare against.
 - If given a block, the `confine` is suitable if the block returns a value other than `nil` or `false`.
3. An optional `has_weight` statement:
 - When multiple resolutions are available for a fact, resolutions are evaluated from highest weight value to lowest.
 - Must be an integer greater than 0.
 - Defaults to the number of `confine` statements for the resolution.
4. A `setcode` statement that determines the value of the fact:
 - Can take either a string or a block.
 - If given a string, `Facter` executes it as a shell command. If the command succeeds, the output of the command is the value of the fact. If the command fails, the next suitable resolution is evaluated.
 - If given a block, the block’s return value is the value of the fact unless the block returns `nil`. If `nil` is returned, the next suitable resolution is evaluated.
 - Can execute shell commands within a `setcode` block, using the `Facter::Core::Execution.exec` function.
 - If multiple `setcode` statements are evaluated for a single resolution, only the last `setcode` block is used.

Note: Set all code inside the sections outlined above — there must not be any code outside `setcode` and `confine` blocks other than an optional `has_weight` statement in a custom fact.

How to format facts

The format of a fact is important because of the way that Factor evaluates them — by reading *all* the fact definitions. If formatted incorrectly, Factor can execute code too early. You need to use the `setcode` correctly. Below is a *good* example and a *bad* example of a fact, showing you where to place the `setcode`.

Good:

```
Factor.add('phi') do
  confine owner: "BTO"
  confine :kernel do |value|
    value == "Linux"
  end

  setcode do
    bar=Factor.value('theta')
    bar + 1
  end
end
```

In this example, the `bar=Factor.value('theta')` call is guarded by `setcode`, which means it is not executed unless or until it is appropriate to do so. Factor loads all `Factor.add` blocks first, use any OS or confine/weight information to decide which facts to evaluate, and once it chooses, it selectively executes `setcode` blocks for each fact that it needs.

Bad:

```
Factor.add('phi') do
  confine owner: "BTO"
  confine :kernel do |value|
    value == "Linux"
  end

  bar = Factor.value('theta')

  setcode do
    bar + 1
  end
end
```

In this example, the `Factor.value('theta')` call is outside of the guarded `setcode` block and in the unguarded part of the `Factor.add` block. This means that the statement always executes, on every system, regardless of confine, weight, or which resolution of `phi` is appropriate. Any code with possible side-effects, or code pertaining to figuring out the value of a fact, must be kept inside the `setcode` block. The only code left outside `setcode` is code that helps Factor choose which resolution of a fact to use.

Examples

The following example shows a minimal fact that relies on a single shell command:

```
Factor.add(:rubypath) do
  setcode 'which ruby'
end
```

The following example shows different resolutions for different operating systems:

```
Factor.add(:rubypath) do
  setcode 'which ruby'
end

Factor.add(:rubypath) do
  confine osfamily: "Windows"
```

```
# Windows uses 'where' instead of 'which'
setcode 'where ruby'
end
```

The following example shows a more complex fact, confined to Linux with a block:

```
Facter.add(:jruby_installed) do
  confine :kernel do |value|
    value == "Linux"
  end

  setcode do
    # If jruby is present, return true. Otherwise, return false.
    Facter::Core::Execution.which('jruby') != nil
  end
end
```

Writing structured facts

Structured facts can take the form of hashes or arrays.

You don't have to do anything special to mark the fact as structured — if your fact returns a hash or array, Facter recognizes it as a structured fact. Structured facts can have simple or aggregate resolutions.

Example: Returning an array of network interfaces

```
Facter.add(:interfaces_array) do
  setcode do
    interfaces = Facter.value(:interfaces)
    # the 'interfaces' fact returns a single comma-delimited string, such as
    "lo0,eth0,eth1"
    # this splits the value into an array of interface names
    interfaces.split(',')
  end
end
```

Example: Returning a hash of network interfaces to IP addresses

```
Facter.add(:interfaces_hash) do
  setcode do
    interfaces_hash = {}

    Facter.value(:interfaces_array).each do |interface|
      ipaddress = Facter.value("ipaddress_#{interface}")
      if ipaddress
        interfaces_hash[interface] = ipaddress
      end
    end

    interfaces_hash
  end
end
```

Writing facts with aggregate resolutions

Aggregate resolutions allow you to split up the resolution of a fact into separate chunks.

By default, Facter merges hashes with hashes or arrays with arrays, resulting in a structured fact, but you can also aggregate the chunks into a flat fact using concatenation, addition, or any other function that you can express in Ruby code.

Main components of aggregate resolutions

Aggregate resolutions have two key differences compared to simple resolutions: the presence of `chunk` statements and the lack of a `setcode` statement. The `aggregate` block is optional, and without it `Facter` merges hashes with hashes or arrays with arrays.

1. A call to `Facter.add(:fact_name, :type => :aggregate):`
 - Introduces a new fact or a new resolution for an existing fact with the same name.
 - The name can be either a symbol or a string.
 - The `:type => :aggregate` parameter is required for aggregate resolutions.
 - The rest of the fact is wrapped in the `add` call's `do ... end` block.
2. Zero or more `confine` statements:
 - Determine whether the resolution is suitable and (therefore is evaluated).
 - They can either match against the value of another fact or evaluate a Ruby block.
 - If given a symbol or string representing a fact name, a block is required and the block receives the fact's value as an argument.
 - If given a hash, the keys are expected to be fact names. The values of the hash are either the expected fact values or an array of values to compare against.
 - If given a block, the `confine` is suitable if the block returns a value other than `nil` or `false`.
3. An optional `has_weight` statement:
 - Evaluates multiple resolutions for a fact from highest weight value to lowest.
 - Must be an integer greater than 0.
 - Defaults to the number of `confine` statements for the resolution.
4. One or more calls to `chunk`, each containing:
 - A name (as the argument to `chunk`).
 - A block of code, which is responsible for resolving the chunk to a value. The block's return value is the value of the chunk; it can be any type, but is typically a hash or array.
5. An optional `aggregate` block:
 - If absent, `Facter` automatically merges hashes with hashes or arrays with arrays.
 - To merge the chunks in any other way, you need to make a call to `aggregate`, which takes a block of code.
 - The block is passed one argument (`chunks`, in the example), which is a hash of chunk name to chunk value for all the chunks in the resolution.

Example: Building a structured fact progressively

This example builds a new fact, `networking_primary_sha`, by progressively merging two chunks. One chunk encodes each networking interface's MAC address as an encoded base64 value, and the other determines if each interface is the system's primary interface.

```
require 'digest'
require 'base64'

Facter.add(:networking_primary_sha, :type => :aggregate) do

  chunk(:sha256) do
    interfaces = {}

    Facter.value(:networking)['interfaces'].each do |interface, values|
      if values['mac']
        hash = Digest::SHA256.digest(values['mac'])
        encoded = Base64.encode64(hash)
        interfaces[interface] = {:mac_sha256 => encoded.strip}
      end
    end
  end
end
```

```

    interfaces
  end

  chunk(:primary?) do
    interfaces = {}

    Facter.value(:networking)['interfaces'].each do |interface, values|
      interfaces[interface] = {:primary? => (interface ==
Facter.value(:networking)['primary'])}
    end

    interfaces
  end
  # Facter merges the return values for the two chunks
  # automatically, so there's no aggregate statement.
end

```

The fact's output is organized by network interface into hashes, each containing the two chunks:

```

{
  bridge0 => {
    mac_sha256 => "bfgEFV7m1V04HYU6UqzoNoVmnPIEKWRSUOU650j0Wkk=",
    primary?   => false
  },
  en0 => {
    mac_sha256 => "6Fd3Ws2z+aIl8vNmClCbzxiO2TddyFBChMlIU+QB28c=",
    primary?   => true
  },
  ...
}

```

Example: Building a flat fact progressively with addition

```

Facter.add(:total_free_memory_mb, :type => :aggregate) do
  chunk(:physical_memory) do
    Facter.value(:memoryfree_mb)
  end

  chunk(:virtual_memory) do
    Facter.value(:swapfree_mb)
  end

  aggregate do |chunks|
    # The return value for this block determines the value of the fact.
    sum = 0
    chunks.each_value do |i|
      sum += i
    end

    sum
  end
end

```

Example: Custom facts with dot notation

Facter 4 introduces a new way to write aggregate custom facts by using dot notation in a fact name. Facter automatically combines the two facts and creates a structured fact. The advantage of this method, is that if one fact throws an exception, and the rest of the facts resolve successfully, Facter still creates a hierarchy with the facts that got resolved. For example:

```

Facter.add('myorg.my_group_1.fact1') do

```

```

    setcode do
      'fact1_value'
    end
  end

  Facter.add('myorg.my_group_1.fact2') do
    setcode do
      'fact2_value'
    end
  end
end

```

Results to:

```

myorg => {
  my_group_1 => {
    fact1 => "fact1_value",
    fact2 => "fact2_value"
  }
}

```

External facts

External facts provide a way to use arbitrary executables or scripts as facts, or set facts statically with structured data. With this information, you can write a custom fact in Perl, C, or a one-line text file.

Executable facts on Unix

Executable facts on Unix work by dropping an executable file into the standard external fact path. A shebang (`#!`) is always required for executable facts on Unix. If the shebang is missing, the execution of the fact fails.

An example external fact written in Python:

```

#!/usr/bin/env python
data = {"key1" : "value1", "key2" : "value2" }

for k in data:
    print "%s=%s" % (k,data[k])

```

You must ensure that the script has its execute bit set:

```

chmod +x /etc/facter/facts.d/my_fact_script.py

```

For Facter to parse the output, the script should return key-value pairs, JSON, or YAML.

Custom executable external facts can return data in YAML or JSON format, and Facter parses it into a structured fact. If the returned value is not YAML, Facter falls back to parsing it as a key-value pair.

By using the key-value pairs on STDOUT format, a single script can return multiple facts:

```

key1=value1
key2=value2
key3=value3

```

Executable facts on Windows

Executable facts on Windows work by dropping an executable file into the external fact path. The external facts interface expects Windows scripts to end with a known extension. Line endings can be either LF or CRLF. The following extensions are supported:

- `.com` and `.exe`: binary executables
- `.bat` and `.cmd`: batch scripts

- `.ps1`: PowerShell scripts

The script should return key-value pairs, JSON, or YAML.

Custom executable external facts can return data in YAML or JSON format, and Facter parses it into a structured fact. If the returned value is not YAML, Facter falls back to parsing it as a key-value pair.

By using the key-value pairs on STDOUT format, a single script can return multiple facts:

```
key1=value1
key2=value2
key3=value3
```

Using this format, a single script can return multiple facts in one return.

For batch scripts, the file encoding for the `.bat` or `.cmd` files must be ANSI or UTF8 without BOM.

Here is a sample batch script which outputs facts using the required format:

```
@echo off
echo key1=val1
echo key2=val2
echo key3=val3
REM Invalid - echo 'key4=val4'
REM Invalid - echo "key5=val5"
```

For PowerShell scripts, the encoding used with `.ps1` files is flexible. PowerShell determines the encoding of the file at run time.

Here is a sample PowerShell script which outputs facts using the required format:

```
Write-Host "key1=val1"
Write-Host 'key2=val2'
Write-Host key3=val3
```

Save and execute this PowerShell script on the command line.

Executable fact locations

Distribute external executable facts with `pluginsync`. To add external executable facts to your Puppet modules, place them in `<MODULEPATH>/<MODULE>/facts.d/`.

If you're not using `pluginsync`, then external facts must go in a standard directory. The location of this directory varies depending on your operating system, whether your deployment uses Puppet Enterprise or open source releases, and whether you are running as root or Administrator. When calling Facter from the command line, you can specify the external facts directory with the `--external-dir` option.

Note: These directories don't necessarily exist by default; you might need to create them. If you create the directory, make sure to restrict access so that only administrators can write to the directory.

In a module (recommended):

```
<MODULEPATH>/<MODULE>/facts.d/
```

On Unix, Linux, or Mac OS X, there are three directories:

```
/opt/puppetlabs/facter/facts.d/
/etc/puppetlabs/facter/facts.d/
/etc/facter/facts.d/
```

On Windows:

```
C:\ProgramData\PuppetLabs\facter\facts.d\
```

When running as a non-root or non-Administrator user:

```
<HOME DIRECTORY>/.facter/facts.d/
```

Note: You can use custom facts as a non-root user only if you have first [configured non-root user access](#) and previously run Puppet agent as that same user.

Structured data facts

Facter can parse structured data files stored in the external facts directory and set facts based on their contents.

Structured data files must use one of the supported data types and must have the correct file extension. Facter supports the following extensions and data types:

- `.yaml`: YAML data, in the following format:

```
---
key1: val1
key2: val2
key3: val3
```

- `.json`: JSON data, in the following format:

```
{ "key1": "val1", "key2": "val2", "key3": "val3" }
```

- `.txt`: Key-value pairs, of the `String` data type, in the following format:

```
key1=value1
key2=value2
key3=value3
```

As with executable facts, structured data files can set multiple facts at one time.

```
{
  "datacenter":
  {
    "location": "bfs",
    "workload": "Web Development Pipeline",
    "contact": "Blackbird"
  },
  "provision":
  {
    "birth": "2017-01-01 14:23:34",
    "user": "alex"
  }
}
```

You can also compose multiple external facts in a structured fact using the dot notation. For example:

```
my_org.my_group.my_fact1 = fact1_value
my_org.my_group.my_fact2 = fact2_value
```

Structured data facts on Windows

All of the above types are supported on Windows with the following notes:

- The line endings can be either LF or CRLF.

- The file encoding must be either ANSI or UTF8 without BOM.

Troubleshooting

If your external fact is not appearing in Factor's output, running Factor in debug mode can reveal why and tell you which file is causing the problem:

```
# puppet facts --debug
```

One possible cause is a fact that returns invalid characters. For example if you used a hyphen instead of an equals sign in your script `test.sh`:

```
#!/bin/bash
echo "key1-value1"
```

Running `puppet facts --debug` yields the following message:

```
...
Debug: Factor: resolving facts from executable file "/tmp/test.sh".
Debug: Factor: executing command: /tmp/test.sh
Debug: Factor: key1-value1
Debug: Factor: ignoring line in output: key1-value1
Debug: Factor: process exited with status code 0.
Debug: Factor: completed resolving facts from executable file "/tmp/
test.sh".
...
```

If you find that an external fact does not match what you have configured in your `facts.d` directory, make sure you have not defined the same fact using the external facts capabilities found in the `stdlib` module.

Drawbacks

While external facts provide a mostly-equal way to create variables for Puppet, they have a few drawbacks:

- An external fact cannot internally reference another fact. However, due to parse order, you can reference an external fact from a Ruby fact.
- External executable facts are forked instead of executed within the same process.

Related information

[Custom facts overview](#) on page 224

You can add custom facts by writing snippets of Ruby code on the primary Puppet server. Puppet then uses plug-ins in modules to distribute the facts to the client.

Configuring Factor with `factor.conf`

The `factor.conf` file is a configuration file that allows you to cache and block fact groups and facts, and manage how Factor interacts with your system. There are four sections: `facts`, `global`, `cli` and `fact-groups`. All sections are optional and can be listed in any order within the file.

When you run Factor from the Ruby API, only the `facts` section and limited `global` settings are loaded.

Example `factor.conf` file:

```
facts : {
  blocklist : [ "file system", "EC2", "os.architecture" ],
  ttls : [
    { "timezone" : 30 days },
  ]
}
global : {
```

```

external-dir      : [ "path1", "path2" ],
custom-dir        : [ "custom/path" ],
no-external-facts : false,
no-custom-facts   : false,
no-ruby           : false
}
cli : {
  debug      : false,
  trace      : true,
  verbose    : false,
  log-level  : "warn"
}
fact-groups : {
  custom-group-name : [ "os.name", "ssh" ],
}

```

Note: The `factor.conf` file is in hocon format. Use `//` or `#` for comments.

Location

Factor does not create the `factor.conf` file automatically, so you must create it manually, or use a module to manage it. Factor loads the file by default from `/etc/puppetlabs/factor/factor.conf` on *nix systems and `C:\ProgramData\PuppetLabs\factor\etc\factor.conf` on Windows. Or you can specify a different default with the `--config` command line option:

```
factor --config path/to/my/config/file/factor.conf
```

facts

This section of `factor.conf` contains settings that affect fact groups. A fact group contains one or more individual facts. When you block or cache a fact group, all the facts from that group are affected. You can cache any type of fact, including custom and external.

The settings in this section are:

- `blocklist` — Prevents all facts within the listed groups from being resolved when Factor runs. Use the `--list-block-group` command line option to list valid groups.
- `ttls` — Caches the key-value pairs of groups and their duration to be cached. Use the `--list-cache-group` command line option to list valid groups.
 - Cached facts are stored as JSON in `/opt/puppetlabs/factor/cache/cached_facts` on *nix and `C:\ProgramData\PuppetLabs\factor\cache\cached_facts` on Windows.

Caching and blocking facts is useful when Factor is taking a long time and slowing down your code. When a system has a lot of something — for example, mount points or disks — Factor can take a long time to collect the facts from each one. When this is a problem, you can speed up Factor's collection by either blocking facts you're uninterested in (`blocklist`), or caching ones you don't need retrieved frequently (`ttls`).

To see a list of valid group names, from the command line, run `factor --list-block-groups` or `factor --list-cache-groups`. The output shows the fact group at the top level, with all facts in that group nested below:

```

$ factor --list-block-groups
EC2
- ec2_metadata
- ec2_userdata
file system
- mountpoints
- filesystems
- partitions

```

If you want to block any of these groups, add the group name to the `facts` section of `factor.conf`, with the `blocklist` setting:

```
facts : {
  blocklist : [ "file system" ],
}
```

Here, the `file system` group has been added, so the `mountpoints`, `filesystems`, and `partitions` facts are now prevented from loading.

You can also specify fact names with the `blocklist` and `ttls` settings. For example:

```
facts : {
  blocklist : [ "my_external_fact.txt", "my_executable_fact.sh" ],
}
```

```
ttls : [
  { "timezone" : 30 days },
  { "networking.hostname" : 6 hours },
]
```

global

The `global` section of `factor.conf` contains settings to control how Factor interacts with its external elements on your system.

Setting	Effect	Default
<code>external-dir</code>	A list of directories to search for external facts.	
<code>custom-dir</code>	A list of directories to search for custom facts.	
<code>no-external*</code>	If <code>true</code> , prevents Factor from searching for external facts.	<code>false</code>
<code>no-custom*</code>	If <code>true</code> , prevents Factor from searching for custom facts.	<code>false</code>
<code>no-ruby*</code>	If <code>true</code> , prevents Factor from loading its Ruby functionality.	<code>false</code>

*Not available when you run Factor from the Ruby API.

cli

The `cli` section of `factor.conf` contains settings that affect Factor's command line output. All of these settings are ignored when you run Factor from the Ruby API.

Setting	Effect	Default
<code>debug</code>	If <code>true</code> , Factor outputs debug messages.	<code>false</code>
<code>trace</code>	If <code>true</code> , Factor prints stacktraces from errors arising in your custom facts.	<code>false</code>
<code>verbose</code>	If <code>true</code> , Factor outputs its most detailed messages.	<code>false</code>

Setting	Effect	Default
log-level	Sets the minimum level of message severity that gets logged. Valid options: "none", "fatal", "error", "warn", "info", "debug", "trace".	"warn"

fact-groups

The `fact-groups` group lets you define your own custom groups in `facter.conf`. A group can contain any type of fact, except external facts. For aggregated facts, you can specify the root fact using the dot notation. You can use these defined groups for blocking (`blocklist`) or caching (`ttls`).

```
fact-groups : {
  custom-group-name : [ "networking.hostname", "ssh" ],
}
```

Hiera

Hiera is a built-in key-value configuration data lookup system, used for separating data from Puppet code.

- [About Hiera](#) on page 241

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.

- [Getting started with Hiera](#) on page 245

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.

- [Configuring Hiera](#) on page 249

The Hiera configuration file is called `hiera.yaml`. It configures the hierarchy for a given layer of data.

- [Creating and editing data](#) on page 256

Important aspects of using Hiera are merge behavior and interpolation.

- [Looking up data with Hiera](#) on page 264

- [Writing new data backends](#) on page 269

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.

- [Upgrading to Hiera 5](#) on page 276

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.

About Hiera

Puppet's strength is in reusable code. Code that serves many needs must be configurable: put site-specific information in external configuration data files, rather than in the code itself.

Puppet uses Hiera to do two things:

- Store the configuration data in key-value pairs
- Look up what data a particular module needs for a given node during catalog compilation

This is done via:

- Automatic Parameter Lookup for classes included in the catalog
- Explicit lookup calls

Hiera's hierarchical lookups follow a “defaults, with overrides” pattern, meaning you specify common data one time, and override it in situations where the default won't work. Hiera uses Puppet's facts to specify data sources, so you can structure your overrides to suit your infrastructure. While using facts for this purpose is common, data-sources can also be defined without the use of facts.

Puppet 5 comes with support for JSON, YAML, and EYAML files.

Related topics: [Automatic Parameter Lookup](#).

Hiera hierarchies

Hiera looks up data by following a hierarchy — an ordered list of data sources.

Hierarchies are configured in a `hiera.yaml` configuration file. Each level of the hierarchy tells Hiera how to access some kind of data source. A hierarchy is usually organized like this:

```
---
version: 5
defaults: # Used for any hierarchy level that omits these keys.
  datadir: data # This path is relative to hiera.yaml's directory.
  data_hash: yaml_data # Use the built-in YAML backend.

hierarchy:
  - name: "Per-node data" # Human-readable name.
    path: "nodes/{trusted.certname}.yaml" # File path, relative to
    datadir. # ^^^ IMPORTANT: include the file
    extension!

  - name: "Per-datacenter business group data" # Uses custom facts.
    path: "location/{facts.whereami}/{facts.group}.yaml"

  - name: "Global business group data"
    path: "groups/{facts.group}.yaml"

  - name: "Per-datacenter secret data (encrypted)"
    lookup_key: eyaml_lookup_key # Uses non-default backend.
    path: "secrets/{facts.whereami}.eyaml"
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
      pkcs7_public_key: /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem

  - name: "Per-OS defaults"
    path: "os/{facts.os.family}.yaml"

  - name: "Common data"
    path: "common.yaml"
```

In this example, every level configures the path to a YAML file on disk.

Hierarchies interpolate variables

Most levels of a hierarchy interpolate variables into their configuration:

```
path: "os/{facts.os.family}.yaml"
```

The percent-and-braces `{variable}` syntax is a Hiera interpolation token. It is similar to the Puppet language's `${expression}` interpolation tokens. Wherever you use an interpolation token, Hiera determines the variable's value and inserts it into the hierarchy.

The `facts.os.family` uses the Hiera special key `.subkey` notation for accessing elements of hashes and arrays. It is equivalent to `$facts['os']['family']` in the Puppet language but the 'dot' notation produces an

empty string instead of raising an error if parts of the data is missing. Make sure that an empty interpolation does not end up matching an unintended path.

You can only interpolate values into certain parts of the config file. For more info, see the `hiera.yaml` format reference.

With node-specific variables, each node gets a customized set of paths to data. The hierarchy is always the same.

Hiera searches the hierarchy in order

After Hiera replaces the variables to make a list of concrete data sources, it checks those data sources in the order they were written.

Generally, if a data source doesn't exist, or doesn't specify a value for the current key, Hiera skips it and moves on to the next source, until it finds one that exists — then it uses it. Note that this is the default merge strategy, but does not always apply, for example, Hiera can use data from all data sources and merge the result.

Earlier data sources have priority over later ones. In the example above, the node-specific data has the highest priority, and can override data from any other level. Business group data is separated into local and global sources, with the local one overriding the global one. Common data used by all nodes always goes last.

That's how Hiera's "defaults, with overrides" approach to data works — you specify common data at lower levels of the hierarchy, and override it at higher levels for groups of nodes with special needs.

Layered hierarchies

Hiera uses layers of data with a `hiera.yaml` for each layer.

Each layer can configure its own independent hierarchy. Before a lookup, Hiera combines them into a single super-hierarchy: `global # environment # module`.

Note: There is a fourth layer - `default_hierarchy` - that can be used in a module's `hiera.yaml`. It only comes into effect when there is no data for a key in any of the other regular hierarchies

Assume the example above is an environment hierarchy (in the production environment). If we also had the following global hierarchy:

```
---
version: 5
hierarchy:
  - name: "Data exported from our old self-service config tool"
    path: "selfserve/{trusted.certname}.json"
    data_hash: json_data
    datadir: data
```

And the NTP module had the following hierarchy for default data:

```
---
version: 5
hierarchy:
  - name: "OS values"
    path: "os/{facts.os.name}.yaml"
  - name: "Common values"
    path: "common.yaml"
defaults:
  data_hash: yaml_data
  datadir: data
```

Then in a lookup for the `ntp::servers` key, `thrush.example.com` would use the following combined hierarchy:

- `<CODEDIR>/data/selfserve/thrush.example.com.json`
- `<CODEDIR>/environments/production/data/nodes/thrush.example.com.yaml`

- `<CODEDIR>/environments/production/data/location/belfast/ops.yaml`
- `<CODEDIR>/environments/production/data/groups/ops.yaml`
- `<CODEDIR>/environments/production/data/os/Debian.yaml`
- `<CODEDIR>/environments/production/data/common.yaml`
- `<CODEDIR>/environments/production/modules/ntp/data/os/Ubuntu.yaml`
- `<CODEDIR>/environments/production/modules/ntp/data/common.yaml`

The combined hierarchy works the same way as a layer hierarchy. Hiera skips empty data sources, and either returns the first found value or merges all found values.

Note: By default, `datadir` refers to the directory named ‘data’ next to the `hiera.yaml`.

Tips for making a good hierarchy

- Make a short hierarchy. Data files are easier to work with.
- Use the roles and profiles method to manage less data in Hieria. 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. Hieria searches every data source in the global layer’s hierarchy before checking any source in the environment layer.

The global layer

The configuration file for the global layer is located, by default, in `$confdir/hiera.yaml`. You can change the location by changing the `hiera_config` setting in `puppet.conf`.

Hiera has one global hierarchy. Because it goes before the environment layer, it’s useful for temporary overrides, for example, when your ops team needs to bypass its normal change processes.

The global layer is the only place where legacy Hieria 3 backends can be used - it’s an important piece of the transition period when you migrate you backends to support Hieria 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. Specify all normal data 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 Hieria 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`, use the `default_hierarchy`, as those bindings are excluded from merges. When placed in the regular hierarchy in the module's hierarchy the bindings are merged when a merge lookup is performed.

It comes last in Hieradata'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
<code>ntp::servers</code>	<code>ntp</code>
<code>jenkins::port</code>	<code>jenkins</code>
<code>secure_server</code>	<code>(none)</code>

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 244

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 256

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 249

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 Hieras: a defaults-with-overrides pattern for data lookup, using a node-specific list of data sources.

Related information

[Interpolation](#) on page 261

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

[Hiera hierarchies](#) on page 242

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 roles and profile method](#) 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 to 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 Hieras, 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 these keys in your Hiera data. Depending on where in your hierarchy you want to set the parameters, you can add them to your [common data](#), [os data](#), or [per-node 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 265

The lookup function uses Hiera to retrieve a value for a given key.

Write data: Set values in common data

Set values in your common data — the level at the bottom of your hierarchy.

This hierarchy level uses the YAML backend for data, which means the data goes into a YAML file. To know where to put that file, combine the following pieces of information:

- The current environment's directory.
- The data directory, which is a subdirectory of the environment. By default, it's <ENVIRONMENT>/data.
- The file path specified by the hierarchy level.

In this case, /etc/puppetlabs/code/environments/production/ + data/ + common.yaml.

Open that YAML file in an editor, and set values for two of the class's parameters.

```
# /etc/puppetlabs/code/environments/production/data/common.yaml
---
profile::hiera_test::ssl: false
profile::hiera_test::backups_enabled: true
```

The third parameter, \$site_alias, has a default value defined in code, so you can omit it from the data.

Write data: Set per-operating system data

The second level of the hierarchy uses the `os` fact to locate its data file. This means it can use different data files depending on the operating system of the current node.

For this example, suppose that your developers use MacBook laptops, which have an OS family of Darwin. If a developer is running an app instance on their laptop, it should not send data to your production backup server, so set \$backups_enabled to false.

If you do not run Puppet on any Mac laptops, choose an OS family that is meaningful to your infrastructure.

1. Locate the data file, by replacing `%{facts.os.family}` with the value you are targeting:

```
/etc/puppetlabs/code/environments/production/data/ + os/ + Darwin + .yaml
```

2. Add the following contents:

```
# /etc/puppetlabs/code/environments/production/data/os/Darwin.yaml
---
profile::hiera_test::backups_enabled: false
```

3. Compile to observe that the override takes effect.

Related topics: [the os fact](#).

Write data: Set per-node data

The highest level of the example hierarchy uses the value of `$trusted['certname']` to locate its data file, so you can set data by name for each individual node.

This example supposes you have a server named `jenkins-prod-03.example.com`, and configures it to use SSL and to serve this application at the hostname `ci.example.com`. To try this out, choose the name of a real server that you can run Puppet on.

1. To locate the data file, replace `%{trusted.certname}` with the node name you're targeting:

```
/etc/puppetlabs/code/environments/production/data/ + nodes/ + jenkins-
prod-03.example.com + .yaml
```

2. Open that file in an editor and add the following contents:

```
# /etc/puppetlabs/code/environments/production/data/nodes/jenkins-
prod-03.example.com.yaml
---
profile::hiera_test::ssl: true
profile::hiera_test::site_alias: ci.example.com
```

3. Compile to observe that the override takes effect.

Related topics: [\\$trusted\['certname'\]](#).

Testing Hiera data on the command line

As you set Hiera data or rearrange your hierarchy, it is important to double-check the data a node receives.

The `puppet lookup` command helps test data interactively. For example:

```
puppet lookup profile::hiera_test::backups_enabled --environment production
--node jenkins-prod-03.example.com
```

This returns the value `true`.

To use the `puppet lookup` command effectively:

- Run the command on a Puppet Server node, or on another node that has access to a full copy of your Puppet code and configuration.
- The node you are testing against should have contacted the server at least one time as this makes the facts for that node available to the `lookup` command (otherwise you need to supply the facts yourself on the command line).
- Make sure the command uses the global `confdir` and `codedir`, so it has access to your live data. If you're not running `puppet lookup` as root user, specify `--codedir` and `--confdir` on the command line.
- If you use PuppetDB, you can use any node's facts in a lookup by specifying `--node <NAME>`. Hiera can automatically get that node's real facts and use them to resolve data.
- If you do not use PuppetDB, or if you want to test for a set of facts that don't exist, provide facts in a YAML or JSON file and specify that file as part of the command with `--facts <FILE>`. To get a file full of facts, rather than creating one from scratch, run `facter -p --json > facts.json` on a node that is similar to the node you want to examine, copy the `facts.json` file to your Puppet Server node, and edit it as needed.
 - Puppet Development Kit comes with predefined fact sets for a variety of platforms. You can use those if you want to test against platforms you do not have, or if you want "typical facts" for a kind of platform.
- If you are not getting the values you expect, try re-running the command with `--explain`. The `--explain` flag makes Hiera output a full explanation of which data sources it searched and what it found in them.

Related topics: [The puppet lookup command](#), [confdir](#), [codedir](#).

Configuring Hiera

The Hiera configuration file is called `hiera.yaml`. It configures the hierarchy for a given layer of data.

Related information

[Hiera configuration layers](#) on page 244

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 242

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. Hieria 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	<code>\$confdir/hiera.yaml</code>	<code>/etc/puppetlabs/puppet/hiera.yaml</code> <code>C:\ProgramData\PuppetLabs\puppet\etc\hiera.yaml</code>
Environment	<code><ENVIRONMENT>/hiera.yaml</code>	<code>/etc/puppetlabs/code/environments/production/hiera.yaml</code> <code>C:\ProgramData\PuppetLabs\code\environments\production\hiera.yaml</code>
Module	<code><MODULE>/hiera.yaml</code>	<code>/etc/puppetlabs/code/environments/production/modules/ntp/hiera.yaml</code> <code>C:\ProgramData\PuppetLabs\code\environments\production\modules\ntp\hiera.yaml</code>

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/nodes/{trusted.certname}.eyaml"
    options:
      pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
      pkcs7_public_key: /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem

  - name: "Per-OS defaults"
    path: "os/{facts.os.family}.yaml"

  - name: "Common data"
    path: "common.yaml"

```

Note: When writing in Hierarchical YAML files, do not use hard tabs for indentation.

The default configuration

If you omit the `hierarchy` or `defaults` keys, Hierarchical 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 Hierarchical 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 Hierarchical 3 backend (global layer only).

For the built-in data providers — YAML, JSON, and HOCON — the key is always `data_hash` and the value is one of `yaml_data`, `json_data`, or `hocon_data`. To set a custom data provider as the default, see the data provider documentation. Whichever key you use, the value must be the name of the custom Puppet function that implements the lookup function.

The hierarchy key

The `hierarchy` key configures the levels of the hierarchy. The value of `hierarchy` must be an array of hashes.

Indent the hash's keys by four spaces, so they line up with the first key. Put an empty line between hashes, to visually distinguish them. For example:

```
hierarchy:
  - name: "Per-node data"
    path: "nodes/{trusted.certname}.yaml"

  - name: "Per-datacenter business group data"
    path: "location/{facts.whereami}/{facts.group}.yaml"
```

The default_hierarchy key

The `default_hierarchy` key is a top-level key. It is initiated when, and only when, the lookup in the regular hierarchy does not find a value. Within this default hierarchy, the normal merging rules apply. The `default_hierarchy` is not permitted in environment or global layers.

If `lookup_options` is used, the values found in the regular hierarchy have no effect on the values found in the `default_hierarchy`, and vice versa. A merge parameter, given in a call to `lookup`, is only used in the regular hierarchy. It does not affect how a value in the default hierarchy is assembled. The only way to influence that, is to use `lookup_options`, found in the default hierarchy.

For more information about the YAML file, see [YAML](#).

Related information

[Hiera hierarchies](#) on page 242

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.
 - `mapped_paths` does not support `glob` expansion.
- `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.
 - `datadir` supports variable interpolation.

For more information on built-in backends, see [YAML](#), [JSON](#), [HOCON](#).

Related information

[Interpolate a Puppet variable](#) on page 262

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
<code>path</code>	String	One file path.
<code>paths</code>	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.
<code>glob</code>	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.
<code>globs</code>	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).
<code>mapped_paths</code>	Array or Hash	A fact that is a collection (array or hash) of values. Hiera expands these values to produce an array of paths.

Note: You can only use one of these keys in a given hierarchy level.

Explicit file extensions are required, for example, `common.yaml`, not `common`.

File paths are relative to the `datadir`: if the full `datadir` is `/etc/puppetlabs/code/environments/production/data` and the file path is set to `"nodes/{trusted.certname}.yaml"`, the full path to the file is `/etc/puppetlabs/code/environments/production/data/nodes/<NODE NAME>.yaml`.

Note: Hierarchy levels should interpolate variables into the path.

Globs are implemented with Ruby's `Dir.glob` method:

- One asterisk (`*`) matches a run of characters.
- Two asterisks (`**`) matches any depth of nested directories.
- A question mark (`?`) matches one character.
- Comma-separated lists in curly braces (`{one,two}`) match any option in the list.
- Sets of characters in square brackets (`[abcd]`) match any character in the set.
- A backslash (`\`) escapes special characters.

Example:

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

The `mapped_paths` key must contain three string elements, in the following order:

- A scope variable that points to a collection of strings.
- The variable name that is mapped to each element of the collection.
- A template where that variable can be used in interpolation expressions.

For example, a fact named `$services` contains the array `["a", "b", "c"]`. The following configuration has the same results as if paths had been specified to be `[service/a/common.yaml, service/b/common.yaml, service/c/common.yaml]`.

```
- name: Example
  mapped_paths: [services, tmp, "service/{tmp}/common.yaml"]
```

Related information

[Interpolation](#) on page 261

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

[The hierarchy](#) on page 246

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, search the plugin's docs for the encryption options. 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 261

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

Configuring a hierarchy level: legacy Hieria 3 backends

If you rely on custom data backends designed for Hieria 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 `hieria.yaml` file.
 - For non-file backends, don't use a file extension.
- `hieria3_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 `hieria.yaml`'s directory: if the config file is at `/etc/puppetlabs/code/environments/production/hieria.yaml` and the `datadir` is set to `data`, the full path to the data directory is `/etc/puppetlabs/code/environments/production/data`. Note that Hieria 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. Hieria converts them to symbols for the backend.

The following example shows roughly equivalent v3 and v5 `hieria.yaml` files using legacy backends:

```
# hieria.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 269

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 261

In Hieradata 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 Hieradata are merge behavior and interpolation.

Set the merge behavior for a lookup

When you look up a key in Hieradata, it is common for multiple data sources to have different values for it. By default, Hieradata 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 Hieradata 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 265

The `lookup` function uses Hieradata 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 for the key, and ignores the rest. This is Hieradata's default behavior.

Specify this merge behavior with one of these:

- `'first'`
- `{ 'strategy' => 'first' }`
- `lookup($key)`
- Nothing (because it's the default)

Unique

A unique merge (also called an array merge) combines any number of array and scalar (string, number, boolean) values to return a merged, flattened array with all matching values for a key. All duplicate values are 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'
- `lookup($key, { 'merge' => 'unique' })`
- `{ 'strategy' => 'unique' }`

Hash

A hash merge combines the keys and values of any number of hashes to return a merged hash of all matching values for a key. Every match must be a hash and the lookup fails if any of the values aren't hashes.

If multiple source hashes have a given key, Hieradata 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, Hieradata 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'
- `lookup($key, { 'merge' => 'hash' })`
- `{ 'strategy' => 'hash' }`

Deep

A deep merge combines the keys and values of any number of hashes to return a merged hash. It contains an array of class names and can be used as a lightweight External Node Classifier (ENC).

If the same key exists in multiple source hashes, Hieradata 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'`
- `include(lookup($key, { 'merge' => 'deep' })))`
- `{ 'strategy' => 'deep', <OPTION> => <VALUE>, ... }` — Adjust the merge behavior with these additional options:
 - `'knockout_prefix'` (String or undef) - Use with a string prefix to indicate a value to remove 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 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 Hieria 3 `hieria_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 `hieria_*` functions is locked to one particular merge behavior. (For example, Hieria only merges first-found, and `hieria_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 Hieradata. 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 Hieradata data sources, including module data, which controls the default merge behavior for other keys in your data. Hieradata 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 hieradata.yaml file**. For example, you can set lookup_options in common.yaml.

Defining Merge Behavior with lookup_options

In your Hieradata 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
```

Hieradata uses the configured merge behaviors for these keys.

Note: The lookup_options settings have no effect if you are using the deprecated hieradata_* 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 Hieradata 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 264

Puppet looks up the values for class parameters in Hiera, using the fully qualified name of the parameter (`myclass::parameter_one`) as a lookup key.

Use a regular expression in `lookup_options`

You can use regular expressions in `lookup_options` to configure merge behavior for many lookup keys at the same time.

A regular expression key such as `^profile::(.*)::users$` sets the merge behavior for `profile::server::users`, `profile::postgresql::users`, `profile::jenkins::server::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 caret). If `^` isn't the first character, Hiera treats it as a literal key name instead of a regular expression.
3. If this data source is in a module, follow `^` with the module's namespace: its full name, plus the `::` namespace separator. For example, all regular expression lookup options in the `ntp` module must start with `^ntp::`. Starting with anything else results in an error.

The merge behavior you set for that pattern applies to all lookup keys that match it. In cases where multiple lookup options could apply to the same key, Hiera resolves the conflict. For example, if there's a literal (not regular expression) option available, Hiera uses it. Otherwise, Hiera uses the first regular expression that matches the lookup key, using the order in which they appear in the module code.

Note: `lookup_options` are assembled with a hash merge, which puts keys from lower priority data sources before those from higher priority sources. To override a module's regular expression configured merge behavior, use the exact same regular expression string in your environment data, so that it replaces the module's value. A slightly different regular expression won't work because the lower-priority regular expression goes first.

Interpolation

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

Hiera uses interpolation in two places:

- Hierarchies: you can interpolate variables into the `path`, `paths`, `glob`, `globs`, `uri`, `uris`, `datadir`, `mapped_paths`, and `options` of a hierarchy level. This lets each node get a customized version of the hierarchy.
- Data: you can use interpolation to avoid repetition. This takes one of two forms:
 - If some value always involves the value of a fact (for example, if you need to specify a mail server and you have one predictably-named mail server per domain), reference the fact directly instead of manually transcribing it.
 - If multiple keys need to share the same value, write it out for one of them and reuse it for the rest with the `lookup` or `alias` interpolation functions. This makes it easier to keep data up to date, as you only need to change a given value in one place.

Interpolation token syntax

Interpolation tokens consist of the following:

- A percent sign (`%`)
- An opening curly brace (`{`)
- One of:
 - A variable name, optionally using `key.subkey` notation to access a specific member of a hash or array.
 - An interpolation function and its argument.
- A closing curly brace (`}`).

For example, `%{trusted.certname}` or `%{alias("users")}`.

Hiera interpolates values of Puppet data types and converts them to strings. Note that the exception to this is when using an alias. If the alias is the only thing present, then its value is not converted.

In YAML files, any string containing an interpolation token must be enclosed in quotation marks.

Note: Unlike the Puppet interpolation tokens, you can't interpolate an arbitrary expression.

Related topics: [Puppet's data types](#), [Puppet's rules for interpolating non-string values](#).

Interpolate a Puppet variable

The most common thing to interpolate is the value of a Puppet top scope variable.

The `facts` hash, `trusted` hash, and `server_facts` hash are the most useful variables to Hiera and behave predictably.

Note: If you have a hierarchy level that needs to reference the name of a node, get the node's name by using `trusted.certname`. To reference a node's environment, use `server_facts.environment`.

Avoid using local variables, namespaced variables from classes (unless the class has already been evaluated), and Hiera-specific pseudo-variables (pseudo-variables are not supported in Hiera 5).

If you are using Hiera 3 pseudo-variables, see [Puppet variables passed to Hiera](#).

Puppet makes facts available in two ways: grouped together in the `facts` hash (`$facts['networking']`), and individually as top-scope variables (`$networking`).

When you use individual fact variables, specify the (empty) top-scope namespace for them, like this:

- `%{::networking}`

Not like this:

- `%{networking}`

Note: The individual fact names aren't protected the way `$facts` is, and local scopes can set unrelated variables with the same names. In most of Puppet, you don't have to worry about unknown scopes overriding your variables, but in Hiera you do.

To interpolate a Puppet variable:

Use the name of the variable, omitting the leading dollar sign (`$`). Use the Hiera `key.subkey` notation to access a member of a data structure. For example, to interpolate the value of `$facts['networking']['domain']` write: `smtpserver: "mail.%{facts.networking.domain}"`

For more information, see [facts](#), [environments](#).

Related information

[Access hash and array elements using a key.subkey notation](#) on page 268

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 Hieradata sources. If you need to use the same value for multiple keys, you can assign the literal value to one key, then call `lookup` to reuse the value elsewhere. You can edit the value in one place to change it everywhere it's used.

For example, suppose your WordPress profile needs a database server, and you're already configuring that hostname in data because the MySQL profile needs it. You could write:

```
# in location/pdx.yaml:
profile::mysql::public_hostname: db-server-01.pdx.example.com

# in location/bfs.yaml:
profile::mysql::public_hostname: db-server-06.belfast.example.com

# in common.yaml:
profile::wordpress::database_server:
  "%{lookup('profile::mysql::public_hostname')}"
```

The value of `profile::wordpress::database_server` is always the same as `profile::mysql::public_hostname`. Even though you wrote the WordPress parameter in the `common.yaml` data, it's location-specific, as the value it references was set in your per-location data files.

The value referenced by the `lookup` function can contain another call to `lookup`; if you accidentally make an infinite loop, Hieradata detects it and fails.

Note: The `lookup` and `hieradata` 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 reuse Hash, Array, Boolean, Integer or String values.

When you interpolate `alias` in a string, Hieradata 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-existent key returns an empty string, not a Hieradata "not found" condition.

The literal function

The `literal` interpolation function lets you escape a literal percent sign (%) in Hieradata 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 function's argument is the name of a variable.

The following two values would be identical:

```
smtpserver: "mail.%{facts.domain}"
smtpserver: "mail.%{scope('facts.domain')}"
```

Using interpolation functions

To use an interpolation function to insert non-variable values, write:

1. The name of the function.
2. An opening parenthesis.
3. One argument to the function, enclosed in single or double quotation marks.
4. Use the opposite of what the enclosing string uses: if it uses single quotation marks, use double quotation marks.
5. A closing parenthesis.

For example:

```
wordpress::database_server: "%{lookup('instances::mysql::public_hostname')}"
```

Note: There must be no spaces between these elements.

Looking up data with Hiera

Automatic lookup of class parameters

Puppet looks up the values for class parameters in Hiera, using the fully qualified name of the parameter (`myclass::parameter_one`) as a lookup key.

Most classes need configuration, and you can specify them as parameters to a class as this looks up the needed data if not directly given when the class is included in a catalog. There are several ways Puppet sets values for class parameters, in this order:

1. If you're doing a resource-like declaration, Puppet uses parameters that are explicitly set (if explicitly setting `undef`, a looked up value or default is used).
2. Puppet uses Hiera, using `<CLASS NAME>::<PARAMETER NAME>` as the lookup key. For example, it looks up `ntp::servers` for the `ntp` class's `$servers` parameter.
3. If a parameter still has no value, Puppet uses the default value from the parameter's default value expression in the class's definition.
4. If any parameters have no value and no default, Puppet fails compilation with an error.

For example, you can set servers for the NTP class like this:

```
# /etc/puppetlabs/code/production/data/nodes/web01.example.com.yaml
---
ntp::servers:
```



```
- time.example.com
- 0.pool.ntp.org
```

The best way to manage this is to use the [roles and profiles](#) method, which allows you to store a smaller amount of 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 wins. The rationale for allowing this is that you might be using the same hash of options multiple times, and you might want to override the production of the default value. A `default_values_hash` wins over the lambda if it has a value for the looked up key.

Arguments accepted by `lookup`:

- `<NAME>` (String or Array) - The name of the key to look up. This can also be an array of keys. If Hiera doesn't find anything for the first key, it tries with the subsequent ones, only resorting to a default value if none of them succeed.
- `<VALUE TYPE>` (data Type) - A data type that must match the retrieved value; if not, the lookup (and catalog compilation) fails. Defaults to `Data` which accepts any normal value.
- `<MERGE BEHAVIOR>` (String or Hash; see [Merge behaviors](#)) - Whether and how to combine multiple values. If present, this overrides any merge behavior specified in the data sources. Defaults to no value; Hiera uses merge behavior from the data sources if present, otherwise it does a first-found lookup.
- `<DEFAULT VALUE>` (any normal value) - If present, lookup returns this when it can't find a normal value. Default values are never merged with found values. Like a normal value, the default must match the value type. Defaults to no value; if Hiera can't find a normal value, the lookup (and compilation) fails.

- `<OPTIONS HASH>` (Hash) - Alternate way to set the arguments above, plus some less common additional options. If you pass an options hash, you can't combine it with any regular arguments (except `<NAME>`). An options hash can have the following keys:
 - `'name'` - Same as `<NAME>` (argument 1). You can pass this as an argument or in the hash, but not both.
 - `'value_type'` - Same as `<VALUE TYPE>`.
 - `'merge'` - Same as `<MERGE BEHAVIOR>`.
 - `'default_value'` - Same as `<DEFAULT VALUE>`.
 - `'default_values_hash'` (Hash) - A hash of lookup keys and default values. If Hiera can't find a normal value, it checks this hash for the requested key before giving up. You can combine this with `default_value` or a lambda, which is used if the key isn't present in this hash. Defaults to an empty hash.
 - `'override'` (Hash) - A hash of lookup keys and override values. Puppet checks for the requested key in the overrides hash first. If found, it returns that value as the final value, ignoring merge behavior. Defaults to an empty hash.
 - `lookup` - can take a lambda, which must accept a single parameter. This is yet another way to set a default value for the lookup; if no results are found, Puppet passes the requested key to the lambda and use its result as the default value.

Merge behaviors

Hiera uses a hierarchy of data sources, and a given key can have values in multiple sources. Hiera can either return the first value it finds, or continue to search and merge all the values together. When Hiera searches, it first searches the global layer, then the environment layer, and finally the module layer — where it only searches in modules that have a matching namespace. By default (unless you use one of the merge strategies) it is priority/"first found wins", in which case the search ends as soon as a value is found.

Note: Data sources can use the `lookup_options` metadata key to request a specific merge behavior for a key. The lookup function uses that requested behavior unless you specify one.

Examples:

Default values for a lookup:

(Still works, but deprecated)

```
hiera('some::key', 'the default value')
```

(Recommended)

```
lookup('some::key', undef, undef, 'the default value')
```

Look up a key and returning the first value found:

```
lookup('ntp::service_name')
```

A unique merge lookup of class names, then adding all of those classes to the catalog:

```
lookup('classes', Array[String], 'unique').include
```

A deep hash merge lookup of user data, but letting higher priority sources remove values by prefixing them with:

```
lookup( { 'name' => 'users',
          'merge' => {
            'strategy' => 'deep',
            'knockout_prefix' => '--',
          },
        })
```

The puppet lookup command

The puppet lookup command is the command line interface (CLI) for Puppet's lookup function.

The puppet lookup command lets you do Hiera lookups from the command line. You must run it on a node that has a copy of your Hiera data. You can log into a Puppet Server node and run puppet lookup with sudo.

The most common version of this command is:

```
puppet lookup <KEY> --node <NAME> --environment <ENV> --explain
```

The puppet lookup command searches your Hiera data and returns a value for the requested lookup key, so you can test and explore your data. It replaces the hiera command. Hiera relies on a node's facts to locate the relevant data sources. By default, puppet lookup uses facts from the node you run the command on, but you can get data for any other node with the --node NAME option. If possible, the lookup command uses the requested node's real stored facts from PuppetDB. If PuppetDB is not configured or you want to provide other fact values, pass facts from a JSON or YAML file with the --facts FILE option.

Note: The puppet lookup command replaces the hiera command.

Examples

To look up key_name using the Puppet Server node's facts:

```
$ puppet lookup key_name
```

To look up key_name with agent.local's facts:

```
$ puppet lookup --node agent.local key_name
```

To get the first value found for key_name_one and key_name_two with agent.local's facts while merging values and knocking out the prefix 'example' while merging:

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

To lookup key_name with agent.local's facts, and return a default value of 0 if nothing is found:

```
puppet lookup --node agent.local --default 0 key_name
```

To see an explanation of how the value for key_name is found, using agent.local facts:

```
puppet lookup --node agent.local --explain key_name
```

Options

The puppet lookup command has the following command options:

- **--help:** Print a usage message.
- **--explain:** Explain the details of how the lookup was performed and where the final value came from, or the reason no value was found. Useful when debugging Hiera data. If --explain isn't specified, lookup exits with 0 if a value was found and 1 if not. With --explain, lookup always exits with 0 unless there is a major error. You can provide multiple lookup keys to this command, but it only returns a value for the first found key, omitting the rest.
- **--node <NODE-NAME>:** Specify which node to look up data for; defaults to the node where the command is run. The purpose of Hiera is to provide different values for different nodes; use specific node facts to explore your data. If the node where you're running this command is configured to talk to PuppetDB, the command uses the requested node's most recent facts. Otherwise, override facts with the '--facts' option.

- `--facts <FILE>`: Specify a JSON or YAML file that contains key-value mappings to override the facts for this lookup. Any facts not specified in this file maintain their original value.
- `--environment <ENV>`: Specify an environment. Different environments can have different Hieradata.
- `--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 Hieradata configures merge behavior in data.)
- `--default <VALUE>`: A value to return if Hieradata 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 Hieradata 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 Hieradata using a `key.subkey` notation.

You can access hash and array elements when doing the following things:

- Interpolating variables into `hieradata.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 Hieradata lookups, for example `lookup` and `alias`.

To access a single member of an array or hash:

Use the name of the value followed by a period (`.`) and a subkey.

- If the value is an array, the subkey must be an integer, for example: `users.0` returns the first entry in the `users` array.
- If the value is a hash, the subkey must be the name of a key in that hash, for example, `facts.os`.
- To access values in nested data structures, you can chain subkeys together. For example, because the value of `facts.system_uptime` is a hash, you can access its `hours` key with `facts.system_uptime.hours`.

Example:

To look up the value of `home` in this data:

```
accounts::users:
  ubuntu:
    home: '/var/local/home/ubuntu'
```

You would use the following `lookup` command:

```
lookup('accounts::users.ubuntu.home')
```

Hiera dotted notation

The Hiera dotted notation does not support arbitrary expressions for subkeys; only literal keys are valid.

A hash can include literal dots in the text of a key. For example, the value of `$trusted['extensions']` is a hash containing any certificate extensions for a node, but some of its keys can be raw OID strings like `'1.3.6.1.4.1.34380.1.2.1'`. You can access those values in Hiera with the `key.subkey` notation, but you must put quotation marks — single or double — around the affected subkey. If the entire compound key is quoted (for example, as required by the lookup interpolation function), use the other kind of quote for the subkey, and escape quotes (as needed by your data file format) to ensure that you don't prematurely terminate the whole string.

For example:

```
aliased_key: "%{lookup('other_key.\"dotted.subkey\"')}"
# Or:
aliased_key: "%{lookup(\"other_key.'dotted.subkey'\"')}"
```

Note: Using extra quotes prevents digging into dotted keys. For example, if the lookup key contains a dot (.) then the entire key must be enclosed within single quotes within double quotes, for example, `lookup(" 'has.dot' ")`.

Writing new data backends

You can extend Hiera to look up values in data sources, for example, a PostgreSQL database table, a custom web app, or a new kind of structured data file.

To teach Hiera how to talk to other data sources, write a custom backend.

Important: Writing a custom backend is an advanced topic. Before proceeding, make sure you really need it. It is also worth asking the puppet-dev mailing list or Slack channel to see whether there is one you can re-use, rather than starting from scratch.

Custom backends overview

A backend is a custom Puppet function that accepts a particular set of arguments and whose return value obeys a particular format. The function can do whatever is necessary to locate its data.

A backend function uses the modern Ruby functions API or the Puppet language. This means you can use different versions of a Hiera backend in different environments, and you can distribute Hiera backends in Puppet modules.

Different types of data have different performance characteristics. To make sure Hiera performs well with every type of data source, it supports three types of backends: `data_hash`, `lookup_key`, and `data_dig`.

`data_hash`

For data sources where it's inexpensive, performance-wise, to read the entire contents at one time, like simple files on disk. We suggest using the `data_hash` backend type if:

- The cache is alive for the duration of one compilation
- The data is small
- The data can be retrieved all at one time
- Most of the data gets used
- The data is static

For more information, see [data_hash backends](#).

`lookup_key`

For data sources where looking up a key is relatively expensive, performance-wise, like an HTTPS API. We suggest using the `lookup_key` backend type if:

- The data set is big, but only a small portion is used
- The result can vary during the compilation

The `hiera-eyaml` backend is a `lookup_key` function, because decryption tends to affect performance; as a given node uses only a subset of the available secrets, it makes sense to decrypt only on-demand.

For more information, see [lookup_key backends](#).

data_dig

For data sources that can access arbitrary elements of hash or array values before passing anything back to Hieradata, 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 Hieradata; normal functions and Puppet code can not use them.

For more information, see [custom Puppet functions](#), [the modern Ruby functions API](#).

data_hash backends

A `data_hash` backend function reads an entire data source at one time, and returns its contents as a hash.

The built-in YAML, JSON, and HOCON backends are all `data_hash` functions. You can view their source on GitHub:

- [yaml_data.rb](#)
- [json_data.rb](#)
- [hocon_data.rb](#)

Arguments

Hieradata calls a `data_hash` function with two arguments:

- A hash of options
 - The options hash contains a `path` when the entry in `hieradata.yaml` is using `path`, `paths`, `glob`, `globs`, or `mapped_paths`, and the backend receives one call per path to an existing file. When the entry in `hieradata.yaml` is using `uri` or `uris`, the options hash has a `uri` key, and the backend function is called one time per given `uri`. When `uri` or `uris` are used, Hieradata 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 Hieradata](#) for more information. Values in the returned hash can include Hieradata interpolation tokens like `%{variable}` or `%{lookup('key')}`; Hieradata 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 `hieradata_eyaml` backend is a `lookup_key` function.

You can view its source on Git at [eyaml_lookup_key.rb](#).

Arguments

Hieradata calls a `lookup_key` function with three arguments:

- A key to look up.
- A hash of options.
- A `Puppet::LookupContext` object.

Return type

The function must either call the context object's `not_found` method, or return a value for the requested key. It can return `undef` as a value.

Puppet language example signature:

```
function mymodule::hieradata_backend(
  Variant[String, Numeric] $key,
  Hash                      $options,
  Puppet::LookupContext    $context,
)
```

Ruby example signature:

```
dispatch :hieradata_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 `lookup_options` key to configure merge behavior for other keys. See [Configuring merge behavior in Hieradata](#) for more information. To support Hieradata 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 261

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

[Hieradata calling conventions for backend functions](#) on page 273

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 `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 268

Access hash and array members in Hiera using a `key.subkey` notation.

Hiera calling conventions for backend functions

Hiera uses the following conventions when calling backend functions.

Hiera calls `data_hash` one time per data source, calls `lookup_key` functions one time per data source for every unique key lookup, and calls `data_dig` functions one time per data source for every unique sequence of key segments.

However, a given hierarchy level can refer to multiple data sources with the `path`, `paths`, `uri`, `uris`, `glob`, and `globs` settings. Hiera handles each hierarchy level as follows:

- If the `path`, `paths`, `glob`, or `globs` settings are used, Hiera determines which files exist and calls the function one time for each. If no files were found, the function is not be called.
- If the `uri` or `uris` settings are used, Hiera calls the function one time per URI.
- If none of those settings are used, Hiera calls the function one time.

Hiera can call a function again for a given data source, if the inputs change. For example, if `hiera.yaml` interpolates a local variable in a file path, Hiera calls the function again for scopes where that variable has a different value. This has a significant performance impact, so you must interpolate only facts, trusted facts, and server facts in the hierarchy.

The options hash

Hierarchy levels are configured in the `hiera.yaml` file. When calling a backend function, Hiera passes a modified version of that configuration as a hash.

The options hash can contain (depending on whether `path`, `glob`, `uri`, or `mapped_paths` have been set) the following keys:

- `path` - The absolute path to a file on disk. It is present only if `path`, `paths`, `glob`, `globs`, or `mapped_paths` is present in the hierarchy. Hiera never calls the function unless the file is present.
- `uri` - A uri that your function can use to locate a data source. It is present only if `uri` or `uris` is present in the hierarchy. Hiera does not verify the URI before passing it to the function.
- Every key from the hierarchy level's `options` setting. List any options your backend requires or accepts. The `path` and `uri` keys are reserved.

Note: If your backend uses data files, use the context object's `cached_file_data` method to read them.

For example, the following hierarchy level in `hiera.yaml` results in several different options hashes, depending on such things as the current node's facts and whether the files exist:

```
- name: "Secret data: per-node, per-datacenter, common"
  lookup_key: eyaml_lookup_key # eyaml backend
  datadir: data
  paths:
    - "secrets/nodes/{trusted.certname}.eyaml"
    - "secrets/location/{facts.whereami}.eyaml"
    - "common.eyaml"
  options:
    pkcs7_private_key: /etc/puppetlabs/puppet/eyaml/private_key.pkcs7.pem
    pkcs7_public_key: /etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem
```

The various hashes would all be similar to this:

```
{
  'path' => '/etc/puppetlabs/code/environments/production/data/secrets/
nodes/web01.example.com.eyaml',
  'pkcs7_private_key' => '/etc/puppetlabs/puppet/eyaml/
private_key.pkcs7.pem',
  'pkcs7_public_key' => '/etc/puppetlabs/puppet/eyaml/public_key.pkcs7.pem'
}
```

In your function's signature, you can validate the options hash by using the Struct data type to restrict its contents. In particular, note you can disable all of the `path`, `paths`, `glob`, and `globs` settings for your backend by disallowing the `path` key in the options hash.

For more information, see [the Struct data type](#).

Related information

[Configure merge behavior in data](#)

[Configuring a hierarchy level: hiera-eyaml](#) on page 253

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 261

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

The Puppet::LookupContext object and methods

To support caching and other backends needs, Hiera provides a `Puppet::LookupContext` object.

In Ruby functions, the context object is a normal Ruby object of class `Puppet::LookupContext`, and you can call methods with standard Ruby syntax, for example `context.not_found`.

In Puppet language functions, the context object appears as the special data type `Puppet::LookupContext`, that has methods attached. You can call the context's methods using Puppet's chained function call syntax with the method name instead of a normal function call syntax, for example, `$context.not_found`. For methods that take a block, use Puppet's lambda syntax (parameters outside block) instead of Ruby's block syntax (parameters inside block).

`not_found()`

Tells Hiera to halt this lookup and move on to the next data source. Call this method when your function cannot find a matching key or a given lookup. This method returns no value.

For `data_hash` backends, return an empty hash. The empty hash results in `not_found`, and prevents further calls to the provider. Missing data sources are not an issue when using `path`, `paths`, `glob`, or `globs`, but are important for backends that locate their own data sources.

For `lookup_key` and `data_dig` backends, use `not_found` when a requested key is not present in the data source or the data source does not exist. Do not return `undef` or `nil` for missing keys, as these are legal values that can be set in data.

`interpolate(value)`

Returns the provided value, but with any Hiera interpolation tokens (`%{variable}` or `%{lookup('key')}`) replaced by their value. This lets you opt-in to allowing Hiera-style interpolation in your backend's data sources. It works recursively on arrays and hashes. Hashes can interpolate into both keys and values.

In `data_hash` backends, support for interpolation is built in, and you do not need to call this method.

In `lookup_key` and `data_dig` backends, call this method if you want to support interpolation.

`environment_name()`

Returns the name of the environment, regardless of layer.

`module_name()`

Returns the name of the module whose `hiera.yaml` called the function. Returns `undef` (in Puppet) or `nil` (in Ruby) if the function was called by the global or environment layer.

`cache(key, value)`

Caches a value, in a per-data-source private cache. It also returns the cached value.

On future lookups in this data source, you can retrieve values by calling `cached_value(key)`. Cached values are immutable, but you can replace the value for an existing key. Cache keys can be anything valid as a key for a Ruby hash, including `nil`.

For example, on its first invocation for a given YAML file, the built-in `eyaml_lookup_key` backend reads the whole file and caches it, and then decrypts only the specific value that was requested. On subsequent lookups into that file, it gets the encrypted value from the cache instead of reading the file from disk again. It also caches decrypted values so that it won't have to decrypt again if the same key is looked up repeatedly.

The cache is useful for storing session keys or connection objects for backends that access a network service.

Each `Puppet::LookupContext` cache lasts for the duration of the current catalog compilation. A node can't access values cached for a previous node.

Hiera creates a separate cache for each combination of inputs for a function call, including inputs like `name` that are configured in `hiera.yaml` but not passed to the function. Each hierarchy level has its own cache, and hierarchy levels that use multiple paths have a separate cache for each path.

If any inputs to a function change, for example, a path interpolates a local variable whose value changes between lookups, Hiera uses a fresh cache.

`cache_all(hash)`

Caches all the key-value pairs from a given hash. Returns `undef` (in Puppet) or `nil` (in Ruby).

`cached_value(key)`

Returns a previously cached value from the per-data-source private cache. Returns `undef` or `nil` if no value with this name has been cached.

`cache_has_key(key)`

Checks whether the cache has a value for a given key yet. Returns `true` or `false`.

`cached_entries()`

Returns everything in the per-data-source cache as an iterable object. The returned object is not a hash. If you want a hash, use `Hash($context.all_cached())` in the Puppet language or `Hash[context.all_cached()]` in Ruby.

`cached_file_data(path)`

Puppet syntax:

```
cached_file_data(path) |content| { ... }
```

Ruby syntax:

```
cached_file_data(path) { |content| ... }
```

For best performance, use this method to read files in Hiera backends.

`cached_file_data(path) { |content| ... }` returns the content of the specified file as a string. If an optional block is provided, it passes the content to the block and returns the block's return value. For example, the built-in JSON backend uses a block to parse JSON and return a hash:

```
context.cached_file_data(path) do |content|
  begin
    JSON.parse(content)
  rescue JSON::ParserError => ex
    # Filename not included in message, so we add it here.
    raise Puppet::DataBinding::LookupError, "Unable to parse (#{path}):
    #{ex.message}"
  end
end
```

```
end
end
```

On repeated access to a given file, Hiera checks whether the file has changed on disk. If it hasn't, Hiera uses cached data instead of reading and parsing the file again.

This method does not use the same `per-data-source` caches as `cache(key, value)` and similar methods. It uses a separate cache that lasts across multiple catalog compilations, and is tied to Puppet Server's environment cache.

Because the cache can outlive a given node's catalog compilation, do not do any node-specific pre-processing (like calling `context.interpolate`) in this method's block.

```
explain() { 'message' }
```

Puppet syntax:

```
explain() || { 'message' }
```

Ruby syntax:

```
explain() { 'message' }
```

In both Puppet and Ruby, the provided block must take zero arguments.

`explain() { 'message' }` adds a message, which appears in debug messages or when using `puppet lookup --explain`. The block provided to this function must return a string.

The `explain` method is useful for complex lookups where a function tries several different things before arriving at the value. The built-in backends do not use the `explain` method, and they still have relatively verbose explanations. This method is for when you need to provide even more detail.

Hiera never executes the `explain` block unless `explain` is enabled.

Upgrading to Hiera 5

Upgrading to Hiera 5 offers some major advantages. A real environment data layer means changes to your hierarchy are now routine and testable, using multiple backends in your hierarchy is easier and you can make a custom backend.

Note: If you're already a Hiera user, you can use your current code with Hiera 5 without any changes to it. Hiera 5 is fully backward-compatible with Hiera 3. You can even start using some Hiera 5 features—like module data—without upgrading anything.

Hiera 5 uses the same built-in data formats as Hiera 3. You don't need to do mass edits of any data files.

Updating your code to take advantage of Hiera 5 features involves the following tasks:

Task	Benefit
Enable the environment layer, by giving each environment its own <code>hiera.yaml</code> 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 (<code>hiera</code> , <code>hiera_array</code> , <code>hiera_hash</code> , and <code>hiera_include</code>) gain full Hiera 5 powers in any migrated environment, only if there is a <code>hiera.yaml</code> in the environment root.
Convert your global <code>hiera.yaml</code> file to the version 5 format.	You can use new Hiera 5 backends at the global layer.
Convert any experimental (version 4) <code>hiera.yaml</code> files to version 5.	Future-proof any environments or modules where you used the experimental version of Puppet lookup.

Task	Benefit
In Puppet code, replace legacy Hieradata functions (<code>hieradata</code> , <code>hieradata_array</code> , <code>hieradata_hash</code> , and <code>hieradata_include</code>) with <code>lookup()</code> .	Future-proof your Puppet code.
Use Hieradata for default data in modules.	Simplify your modules with an elegant alternative to the "params.pp" pattern.

Considerations for hieradata-eyaml users

Upgrade now. In Puppet 4.9.3, we added a built-in hieradata-eyaml backend for Hieradata 5. (It still requires that the `hieradata-eyaml` gem be installed.) See the usage instructions in the `hieradata.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 Hieradata 3 backends with Hieradata 5, but they'll make upgrading more complex, because you can't move legacy data to the environment layer until there's a Hieradata 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 Hieradata 5 backends.

Considerations for custom `data_binding_terminus` users

Upgrade now, and replace it with a Hieradata 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 `hieradata` (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 Hieradata 5. But because we've deprecated that extension point, you have to replace your custom terminus with a Hieradata 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 Hieradata 5 backends.

After you have a Hieradata 5 backend, integrate it into your hierarchies and delete the `data_binding_terminus` setting.

Related information

[The Puppet lookup function](#) on page 265

The `lookup` function uses Hieradata to retrieve a value for a given key.

[Config file syntax](#) on page 249

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

[Writing new data backends](#) on page 269

You can extend Hieradata 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.

[Hieradata configuration layers](#) on page 244

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 Hieria 3 was `<ENVIRONMENT>/hieradata`, and the default in Hieria 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 Hieria 5 `hieria.yaml` file to the environment.

Each environment needs a Hieria config file that works with its existing data. If this is the first environment you're migrating, see converting a version 3 `hieria.yaml` to version 5. Make sure to reference the new `datadir` name. If you've already migrated at least one environment, copy the `hieria.yaml` file from a migrated environment and make changes to it if necessary.

Save the resulting file as `<ENVIRONMENT>/hieria.yaml`. For example, `/etc/puppetlabs/code/environments/production/hieria.yaml`.

8. If any of your data relies on custom backends that have been ported to Hieria 5, install them in the environment. Hieria 5 backends are distributed as Puppet modules, so each environment can use its own version of them.
9. If you use only file-based Hieria 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 Hieria 3 backends, the global layer still needs access to their data, so you need to sort the files: Hieria 5 data moves to the new data directory, and Hieria 3 data stays in the old data directory. When you have Hieria 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 Hieria data](#) on page 278

A key feature in Hieria 5 is per-environment hierarchy configuration. Because you probably store data in each environment, local `hieria.yaml` files are more logical and convenient than a single global hierarchy.

[Configuring a hierarchy level: legacy Hieria 3 backends](#) on page 254

If you rely on custom data backends designed for Hieria 3, you can use them in your global hierarchy. They are not supported at the environment or module layers.

[Config file syntax](#) on page 249

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

[Custom backends overview](#) on page 269

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 `hieria.yaml` to version 5

Hieria 5 supports three versions of the `hieria.yaml` file: version 3, version 4, and version 5. If you've been using Hieria 3, your existing configuration is a version 3 `hieria.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 `hieria.yaml` files for environments.
- Updating your global configuration to support Hieria 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	<code>data_hash: yaml_data</code>
json	<code>data_hash: json_data</code>
eyaml	<code>lookup_key: eyaml_lookup_key</code>

5. Translate the hierarchy.

The version 5 and version 3 hierarchies work differently:

- In version 3, hierarchy levels don't have a backend assigned to them, and Hiera loops through the entire hierarchy for each backend.
- In version 5, each hierarchy level has one designated backend, as well as its own independent configuration for that backend.

Consult the previous values for the `:backends` key and any backend-specific settings.

In the example above, we used `yaml`, `eyaml`, and `mongodb` backends. Your business only uses Mongo for per-node data, and uses `eyaml` for per-group data. The rest of the hierarchy is irrelevant to these backends. You need one Mongo level and one `eyaml` level, but still want all five levels in YAML. This means Hiera consults multiple backends for per-node and per-group data. You want the YAML version of per-node data to be authoritative, so put it before the Mongo version. The `eyaml` data does not overlap with the unencrypted per-group data, so it doesn't matter where you put it. Put it before the YAML levels. When you translate your hierarchy, you have to make the same kinds of investigations and decisions.

6. Remove hierarchy levels that use `calling_module`, `calling_class`, and `calling_class_path`, which were allowed pseudo-variables in Hiera 3. Anything you were doing with these variables is better accomplished by using the module data layer, or by using the `glob` pattern (if the reason for using them was to enable splitting up data into multiple files, and not knowing in advance what they names of those would be)

`Hiera.yaml` version 5 does not support these. Remove hierarchy levels that interpolate them.

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 Hieria 3 backends.

Check to see if the backend's author has published a Hieria 5 update for it. If so, use that; see its documentation for details on how to configure hierarchy levels for it.

If there is no update, use the version 3 backend in a version 5 hierarchy at the global layer — it does not work in the environment layer. Find a Hieria 5 compatible replacement, or write Hieria 5 backends yourself.

For details on how to configure a legacy backend, see [Configuring a hierarchy level \(legacy Hieria 3 backends\)](#) in the `hieria.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 244

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 269

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 255

Hiera supports custom backends.

[Configuring a hierarchy level: hiera-eyaml](#) on page 253

Hiera 5 (Puppet 4.9.3 and later) includes a native interface for the Hiera eyaml extension, which keeps data encrypted on disk but lets Puppet read it during catalog compilation.

Convert an experimental (version 4) hiera.yaml to version 5

If you used the experimental version of Puppet lookup (Hiera 5's predecessor), you might have some version 4 hiera.yaml files in your environments and modules. Hiera 5 can use these, but you need to convert them, especially if you want to use any backends other than YAML or JSON. Version 4 and version 5 formats are similar.

Consider this example of a version 4 hiera.yaml file:

```
# /etc/puppetlabs/code/environments/production/hiera.yaml
---
version: 4
datadir: data
hierarchy:
  - name: "Nodes"
    backend: yaml
    path: "nodes/{trusted.certname}"

  - name: "Exported JSON nodes"
    backend: json
    paths:
      - "nodes/{trusted.certname}"
      - "insecure_nodes/{facts.networking.fqdn}"

  - name: "virtual/{facts.virtual}"
    backend: yaml

  - name: "common"
    backend: yaml
```

To convert to version 5, make the following changes:

1. Change the value of the version key to 5.
2. Add a file extension to every file path — use "common.yaml", not "common".
3. If any hierarchy levels are missing a path, add one. In version 5, path no longer defaults to the value of name
4. If there is a top-level datadir key, change it to a defaults key. Set a default backend. For example:

```
defaults:
  datadir: data
  data_hash: yaml_data
```

5. In each hierarchy level, delete the `backend` key and replace it with a `data_hash` key. (If you set a default backend in the `defaults` key, you can omit it here.)

v4 backend	v5 equivalent
<code>backend: yaml</code>	<code>data_hash: yaml_data</code>
<code>backend: json</code>	<code>data_hash: json_data</code>

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 249

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

[Custom backends overview](#) on page 269

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 Hierar 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 Hierar 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
<code>hiera('secure_server')</code>	<code>lookup('secure_server')</code>
<code>hiera_array('ntp:servers')</code>	<code>lookup('ntp:servers', {merge => unique})</code>
<code>hiera_hash('users')</code>	<code>lookup('users', {merge => hash})</code> or <code>lookup('users', {merge => deep})</code>
<code>hiera_include('classes')</code>	<code>lookup('classes', {merge => unique}).include</code>

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 265

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

[Merge behaviors](#) on page 256

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.

Note: You must fully qualify Hiera variables for modules in your YAML file. See [Module data with YAML data files](#) on page 288.

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'] {
    '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 an 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'] {
    'Debian': {
      { 'ntp::service_name' => 'ntp' }
    }
    default: {
      {}
    }
  }

  # Merge the hashes, overriding the service name if this platform uses a
  # non-standard one:
  $base_params + $os_params
}
```

Note: The hash merge operator (+) is useful in these functions.

After you have a function, tell Hiera to use it by adding it to the module layer `hiera.yaml`. A 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 can split up the key-values so that some are in the hierarchy, and some in the `default_hierarchy`, depending on whether it makes sense to merge a value or not.

Here we add it to the regular hierarchy:

```
# ntp/hiera.yaml
---
version: 5
hierarchy:
  - name: "NTP class parameter defaults"
    data_hash: "ntp::params"
    # We only need one hierarchy level, because one function provides all the
    data.
```

With Hierar-based defaults, you can simplify your module's main classes:

- They do not need to inherit from any other class.
- You do not need to explicitly set a default value with the `=` operator.
- Instead APL comes into effect for each parameter without a given value. In the example, the function `ntp::params` is called to get the default params, and those can then be either overridden or merged, just as with all values in Hierar.

```
# 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 Hierar 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/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 265

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

[Hiera configuration layers](#) on page 244

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.

Environments

Environments are isolated groups of agent nodes.

- [About environments](#) on page 289

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

- [Creating environments](#) on page 291

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

- [Environment isolation](#) on page 294

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 primary server.

A primary server 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](#).

Look up which environment a node is in

If you need to determine which environment a certain node is part of, look it up using the `puppet node find` command.

To look up which environment a node is in, run `puppet node find <node>` on the Puppet Server host node, replacing `<node>` with the node's exact name.

Alternatively, run `puppet node find <node> --render_as json | jq .environment` to render the output as JSON and return only the environment name.

Note: The node name must **exactly** match the name in the node's certificate, including capitalization. By default, a node's name is its fully qualified domain name, but the node's name can be changed by using the `certname` and `node_name_value` settings on the node itself.

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 primary server.

Environment scenarios

The main uses for environments fall into three categories: permanent test environments, temporary test environments, and divided infrastructure.

Permanent test environments

In a permanent test environment, there is a stable group of test nodes where all changes must succeed before they can be merged into the production code. The test nodes are a smaller version of the whole production infrastructure. They are either short-lived cloud instances or longer-lived virtual machines (VMs) in a private cloud. These nodes stay in the test environment for their whole lifespan.

Temporary test environments

In a temporary test environment, you can test a single change or group of changes by checking the changes out of version control into the `$codedir/environments` directory, where it is detected as a new environment. A temporary test environment can either have a descriptive name or use the commit ID from the version that it is based on. Temporary environments are good for testing individual changes, especially if you need to iterate quickly while developing them. When you're done with a temporary environment, you can delete it. The nodes in a temporary environment are short-lived cloud instances or VMs, which are destroyed when the environment ends.

Divided infrastructure

If parts of your infrastructure are managed by different teams that do not need to coordinate their code, you can split them into environments.

Environments limitations

Environments have limitations, including leakage and conflicts with exported resources.

Plugins can leak between environments

Environment leakage occurs when different versions of Ruby files, such as resource types, exist in multiple environments. When these files are loaded on the primary server, 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 primary servers 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 `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 primary Puppet server. Environments are turned on by default.

Environment structure

The structure of an environment follows several conventions.

When you create an environment, you give it the following structure:

- It contains a `modules` directory, which becomes part of the environment's default module path.
- It contains a `manifests` directory, which is the environment's default main manifest.
- If you are using Puppet 5, it can optionally contain a `hiera.yaml` file.
- It can optionally contain an `environment.conf` file, which can locally override configuration settings, including `modulepath` and `manifest`.

Note: Environment names can contain lowercase letters, numbers, and underscores. They must match the following regular expression rule: `\A[a-z0-9_]+\Z`. If you are using Puppet 5, remove the `environment_data_provider` setting.

Environment resources

An environment specifies resources that the primary server uses when compiling catalogs for agent nodes. The `modulepath`, the main manifest, Hieradata, and the config version script, can all be specified in `environment.conf`.

The modulepath

The `modulepath` is the list of directories Puppet loads modules from. By default, Puppet loads modules first from the environment's directory, and second from the primary server's `puppet.conf` file's `basemodulepath` setting, which can be multiple directories. If the modules directory is empty or absent, Puppet only uses modules from directories in the `basemodulepath`.

Related topics: [module path](#).

The main manifest

The main manifest is the starting point for compiling a catalog. Unless you say otherwise in `environment.conf`, an environment uses the global `default_manifest` setting to determine its main manifest. The value of this setting can be an absolute path to a manifest that all environments share, or a relative path to a file or directory inside each environment.

The default value of `default_manifest` is `./manifests` — the environment's own manifests directory. If the file or directory specified by `default_manifest` is empty or absent, Puppet does not fall back to any other manifest. Instead, it behaves as if it is using a blank main manifest. If you specify a value for this setting, the global manifest setting from `puppet.conf` is not be used by an environment.

Related topics: [main manifest](#), [environment.conf](#), [default_manifest setting](#), [puppet.conf](#).

Hiera data

Each environment can use its own Hiera hierarchy and provide its own data.

Related topics: [Hiera config file syntax](#).

The config version script

Puppet automatically adds a config version to every catalog it compiles, as well as to messages in reports. The version is an arbitrary piece of data that can be used to identify catalogs and events. By default, the config version is the time at which the catalog was compiled (as the number of seconds since January 1, 1970).

The environment.conf file

An environment can contain an `environment.conf` file, which can override values for certain settings.

The `environment.conf` file overrides these settings:

- `modulepath`
- `manifest`
- `config_version`
- `environment_timeout`

Related topics: [environment.conf](#)

Create an environment

Create an environment by adding a new directory of configuration data.

1. Inside your code directory, create a directory called `environments`.
2. Inside the `environments` directory, create a directory with the name of your new environment using the structure: `$codedir/environments/`
3. Create a `modules` directory and a `manifests` directory. These two directories contain your Puppet code.
4. Configure a `modulepath`:
 - a) Set `modulepath` in its `environment.conf` file. If you set a value for this setting, the global `modulepath` setting from `puppet.conf` is not used by an environment.
 - b) Check the `modulepath` by specifying the environment when requesting the setting value:

```
$ sudo puppet config print modulepath --section server --environment
test /etc/puppetlabs/code/environments/test/modules:/etc/puppetlabs/code/
modules:/opt/puppetlabs/puppet/modules.
```

Note: In Puppet Enterprise (PE), every environment must include `/opt/puppetlabs/puppet/modules` in its `modulepath`, because PE uses modules in that directory to configure its own infrastructure.

5. Configure a main manifest:
 - a) Set `manifest` in its `environment.conf` file. As with the global `default_manifest` setting, you can specify a relative path (to be resolved within the environment's directory) or an absolute path.
 - b) Lock all environments to a single global manifest with the `disable_per_environment_manifest` setting — preventing any environment setting its own main manifest.
6. To specify an executable script that determines an environment's config version:
 - a) Specify a path to the script in the `config_version` setting in its `environment.conf` file. Puppet runs this script when compiling a catalog for a node in the environment, and uses its output as the config version. If you specify a value here, the global `config_version` setting from `puppet.conf` is not used by an environment.

Note: If you're using a system binary like `git rev-parse`, specify the absolute path to it. If `config_version` is set to a relative path, Puppet looks for the binary in the environment, not in the system's `PATH`.

Related topics: [Deploying environments with r10k](#), [Code Manager control repositories](#), [disable_per_environment_manifest](#)

Assign nodes to environments via an ENC

You can assign agent nodes to environments by using an external node classifier (ENC). By default, all nodes are assigned to a default environment named `production`.

The interface to set the environment for a node is different for each ENC. Some ENCs cannot manage environments. When writing an ENC:

1. Ensure that the environment key is set in the YAML output that the ENC returns. If the environment key isn't set in the ENC's YAML output, the primary server uses the environment requested by the agent.

Note: The value from the ENC is authoritative, if it exists. If the ENC doesn't specify an environment, the node's config value is used.

Related topics: [writing ENCs](#).

Assign nodes to environments via the agent's config file

You can assign agent nodes to environments by using the agent's config file. By default, all nodes are assigned to a default environment named production.

To configure an agent to use an environment:

1. Open the agent's `puppet.conf` file in an editor.
2. Find the `environment` setting in either the agent or main section.
3. Set the value of the `environment` setting to the name of the environment you want the agent to be assigned to.

When that node requests a catalog from the primary server, it requests that environment. If you are using an ENC and it specifies an environment for that node, it overrides whatever is in the config file.

Note: Nodes cannot be assigned to unconfigured environments. If a node is assigned to an environment that does not exist — no directory of that name in any of the environment path directories — the primary server fails to compile its catalog. The one exception to this is if the default production environment does not exist. In this case, the agent successfully retrieves an empty catalog.

Global settings for configuring environments

The settings in the primary server's `puppet.conf` file configure how Puppet finds and uses environments.

environmentpath

The `environmentpath` setting is the list of directories where Puppet looks for environments. The default value for `environmentpath` is `$codedir/environments`. If you have more than one directory, separate them by colons and put them in order of precedence.

In this example, `temp_environments` is searched before `environments`:

```
$codedir/temp_environments:$codedir/environments
```

If environments with the same name exist in both paths, Puppet uses the first environment with that name that it encounters.

Put the `environmentpath` setting in the main section of the `puppet.conf` file.

basemodulepath

The `basemodulepath` setting lists directories of global modules that all environments can access by default. Some modules can be made available to all environments. The `basemodulepath` setting configures the global module directories.

By default, it includes `$codedir/modules` for user-accessible modules and `/opt/puppetlabs/puppet/modules` for system modules.

Add additional directories containing global modules by setting your own value for `basemodulepath`.

Related topics: [modulepath](#).

environment_timeout

The `environment_timeout` setting sets how often the primary server refreshes information about environments. It can be overridden per-environment.

This setting defaults to 0 (caching disabled), which lowers the performance of your primary server but makes it easy for new users to deploy updated Puppet code. After 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 uses the same global manifest for every environment. If an environment specifies a different manifest in `environment.conf`, Puppet does not compile catalogs nodes in that environment, to avoid serving catalogs with potentially wrong contents.

If this setting is set to true, the `default_manifest` value must be an absolute path.

default_manifest

The `default_manifest` setting specifies the main manifest for any environment that doesn't set a manifest value in `environment.conf`. The default value of `default_manifest` is `./manifests` — the environment's own manifests directory.

The value of this setting can be:

- An absolute path to one manifest that all environments share.
- A relative path to a file or directory inside each environment's directory.

Related topics: [default_manifest setting](#).

Configure the environment timeout setting

The `environment_timeout` setting determines how often the primary Puppet server caches the data it loads from an environment. For best performance, change the settings after you have a mature code deployment process.

1. Set `environment_timeout = unlimited` in `puppet.conf`.
2. Change your code deployment process to refresh the primary server 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 primary server, 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 primary servers 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 primary server 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 primary server. 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 primary servers, 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, allow the `resource_types` folder so that r10k does not purge it.

Note: In Puppet Enterprise (PE), environment isolation is provided by Code Manager. Environment isolation is not supported for r10k with PE.

Troubleshoot environment isolation

If the `generate types` command cannot generate certain types, if the type generated has missing or inaccurate information, or if the generation itself has errors or fails, you get a catalog compilation error of “type not found” or “attribute not found.”

1. To fix catalog compilation errors:
 - a) Ensure that your Puppet resource types are correctly implemented. In addition to implementation errors, check for types with title patterns that contain a `proc` or a `lambda`, as these types cannot be generated. Refactor any problem resource types.
 - b) Regenerate the metadata by removing the environment’s `.resource_types` directory and running the `generate types` command again.
 - c) If you continue to get catalog compilation errors, disable environment isolation to help you isolate the error.
2. To disable environment isolation in open source Puppet:
 - a) Remove the `generate types` command from any r10k hooks.
 - b) Remove the `.resource_types` directory.
3. To disable environment isolation in Puppet Enterprise (PE):
 - a) In `/etc/puppetlabs/puppetserver/conf.d/pe-puppet-server.conf`, remove the `pre-commit-hook-commands` setting.
 - b) In Hiera, set `puppet_enterprise::server::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.

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 \(`codedir`\)](#) on page 296

The `codedir` is the main directory for Puppet code and data. It is used by the primary Puppet server 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 Hieradata and configuration.

- [Config directory \(`confdir`\)](#) on page 297

Puppet's `confdir` is the main directory for the Puppet configuration. It contains configuration files and the SSL data.

- [Main manifest directory](#) on page 298

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*.

- [The modulepath](#) on page 299

The primary server 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.

- [SSL directory \(`ssldir`\)](#) on page 301

Puppet stores its certificate infrastructure in the SSL directory (`ssldir`) which has a similar structure on all Puppet nodes, whether they are agent nodes, primary Puppet servers, or the certificate authority (CA) server.

- [Cache directory \(`vardir`\)](#) on page 303

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.

Code and data directory (`codedir`)

The `codedir` is the main directory for Puppet code and data. It is used by the primary Puppet server 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 Hieradata and configuration.

Location

The `codedir` is located in one of the following locations:

- *nix: `/etc/puppetlabs/code`
- *nix non-root users: `~/puppetlabs/etc/code`
- Windows: `%PROGRAMDATA%\PuppetLabs\code` (usually `C:\ProgramData\PuppetLabs\code`)

When Puppet is running as root, as a Windows user with administrator privileges, or as the puppet user, it uses a system-wide codedir. When running as a non-root user, it uses a codedir in that user's home directory.

When running Puppet commands and services as root or puppet, use the system codedir. To use the same codedir as the Puppet agent, or the primary server, run admin commands such as puppet module with sudo.

To configure the location of the codedir, set the `codedir` setting in your puppet.conf file, such as:

```
codedir = /etc/puppetlabs/code
```

Important: Puppet Server doesn't use the codedir setting in puppet.conf, and instead uses the `jruby-puppet.master-code-dir` setting in `puppetserver.conf`. When using a non-default codedir, you must change both settings.

Interpolation of \$codedir

The value of the codedir is discovered before other settings, so you can refer to it in other puppet.conf settings by using the \$codedir variable in the value. For example, the \$codedir variable is used as part of the value for the environmentpath setting:

```
[server]
environmentpath = $codedir/override_environments:$codedir/environments
```

This allows you to avoid absolute paths in your settings and keep your Puppet-related files together.

Contents

The codedir contains environments, including manifests and modules, a global modules directory for all environments, and Hiera data.

The code and data directories are:

- `environments`: Contains alternate versions of the modules and manifests directories, to enable code changes to be tested on smaller sets of nodes before entering production.
- `modules`: The main directory for modules.

Config directory (confdir)

Puppet's confdir is the main directory for the Puppet configuration. It contains configuration files and the SSL data.

Location

The confdir is located in one of the following locations:

- *nix root users: `/etc/puppetlabs/puppet`
- Non-root users: `~/.puppetlabs/etc/puppet`
- Windows: `%PROGRAMDATA%\PuppetLabs\puppet\etc` (usually `C:\ProgramData\PuppetLabs\puppet\etc`)

When Puppet is running as root, a Windows user with administrator privileges, or the puppet user, it uses a system-wide confdir. When running as a non-root user, it uses a confdir in that user's home directory.

When running Puppet commands and services as root or puppet, usually you want to use the system codedir. To use the same codedir as the Puppet agent or the primary Puppet server, run admin commands with sudo.

Puppet's confdir can't be set in the puppet.conf, because Puppet needs the confdir to locate that config file. Instead, run commands with the `--confdir` parameter to specify the confdir. If `--confdir` isn't specified when a Puppet application is started, the command uses the default confdir location.

Puppet Server uses the `jruby-puppet.master-conf-dir` setting in [puppetserver.conf](#) to configure its confdir. If you are using a non-default confdir, you must specify `--confdir` when you run commands like `puppet module` to ensure they use the same directories as Puppet Server.

Interpolation of `$confdir`

The value of the confdir is discovered before other settings, so you can reference it, using the `$confdir` variable, in the value of any other setting in `puppet.conf`.

If you need to set nonstandard values for some settings, using the `$confdir` variable allows you to avoid absolute paths and keep your Puppet-related files together.

Contents

The confdir contains several config files and the SSL data. You can change their locations, but unless you have a technical reason that prevents it, use the default structure. Click the links to see documentation for the files and directories in the codedir.

On all nodes, agent and primary server, 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 primary server nodes, and sometimes standalone nodes that run Puppet apply, the confdir also contains:

- [auth.conf](#): Access control rules for the primary server's network services.
- [fileserver.conf](#): Configuration for additional fileserver mount points.
- [hiera.yaml](#): The global configuration for Hiera data lookup. Environments and modules can also have their own `hiera.yaml` files.

Note: To provide backward compatibility for some existing Puppet 4 installations, if a `hiera.yaml` file exists in the codedir, it takes precedence over `hiera.yaml` in the confdir. To ensure that Puppet honors the configuration in the confdir, remove any `hiera.yaml` file that is present in the codedir.

- [routes.yaml](#): Advanced configuration of indirect behavior.

On certificate authority servers, 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 primary Puppet server

The primary Puppet server 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.conf` `manifest` 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 primary server uses for a given environment, run:

```
puppet config print manifest --section server --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 primary server 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 server --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 primary server. 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 primary server (custom resource types, custom functions).
- Plugins used by `puppet apply`.
- Plugins present in the agent's modulepath (which is usually empty but might not be when running an agent on a node that is also a primary server).

In this case, the plugins are handled on a **per-file** basis instead of per-module. If a duplicate module in a 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 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, primary Puppet servers, or the certificate authority (CA) server.

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 primary server 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 server.

Agent and primary server credentials are identified by [certname](#), so an agent process and a primary server process running on the same server can use the same credentials.

The `ssldir` for the Puppet CA, which runs on the CA server, contains similar credentials: private and public keys, a certificate, and a primary server 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 primary server. 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 server only): Contains the files used by Puppet's certificate authority. Mode: 0755. Setting: `cadir`.
 - `ca_crl.pem`: The primary server 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 primary server 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: `signedir`.
 - `<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 primary servers. 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 `puppetserver ca generate` command. It never contains the private key for the CA certificate. Mode: 0750. Setting: `privatekeydir`.
 - `<certname>.pem`: This node's private key. Mode: 0600. Setting: `hostprivkey`.

- `public_keys` (directory): Contains public keys generated by this node in preparation for generating a CSR. Mode: 0755. Setting: `publickeydir`.
- `<certname>.pem`: This node's public key. Mode: 0644. Setting: `hostpubkey`.

Cache directory (`vardir`)

As part of its normal operations, Puppet generates data which is stored in a cache directory called `vardir`. You can mine the data in `vardir` for analysis, or use it to integrate other tools with Puppet.

Location

The cache directory for Puppet Server defaults to `/opt/puppetlabs/server/data/puppetserver`.

The cache directory for the Puppet agent and Puppet apply can be found at one of the following locations:

- *nix systems: `/opt/puppetlabs/puppet/cache`.
- Non-root users: `~/.puppetlabs/opt/puppet/cache`.
- Windows: `%PROGRAMDATA%\PuppetLabs\puppet\cache` (usually `C:\Program Data\PuppetLabs\puppet\cache`).

When Puppet is running as root, a Windows user with administrator privileges, or the `puppet` user, uses a system-wide cache directory. When running as a non-root user, it uses a cache directory in the user's home directory.

Because you usually run Puppet's commands and services as root or `puppet`, the system cache directory is what you usually want to use.

Important: To use the same directories as the agent or primary server, admin commands like `puppetserver ca`, must run with `sudo`.

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

You can specify Puppet's cache directory on the command line by using the `--vardir` option, but you can't set it in `puppet.conf`. If `--vardir` isn't specified when a Puppet application is started, it uses the default cache directory location.

To configure the Puppet Server cache directory, use the `jruby-puppet.server-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 primary server 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	Puppet uses this as a cache for plugins (custom facts, types and providers, functions) synced from a primary server. Do not change its contents. If you delete it, the plugins are restored on the next Puppet run.
client_data	client_datadir	
clientbucket	clientbucketdir	
client_yaml	clientyamldir	
devices	devicedir	
lib/facter	factpath	
facts	factpath	
facts.d	pluginfactdest	
lib	libdir, plugindest	
puppet-module	module_working_dir	
puppet-module/skeleton	module_skeleton_dir	When the option to store reports is enabled, a primary server stores reports received from agents as YAML files in this directory. You can mine these reports for analysis.
reports	reportdir	
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	This file is useful for external integration. It lists all of the classes assigned to this agent node.
agent_disabled.lock	agent_disabled_lockfile	
classes.txt	classfile	
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.

File or directory name	Config setting	Notes
<code>last_run_summary.yaml</code>	<code>lastrunfile</code>	This file is stored in a public directory and is visible to external monitoring tools — making sure the Puppet agent is running every 30 minutes.
<code>last_run_report.yaml</code>	<code>lastrunreport</code>	
<code>resources.txt</code>	<code>resourcefile</code>	
<code>state.yaml</code>	<code>statefile</code>	

Puppet services and tools

Puppet provides a number of core services and administrative tools to manage systems with or without a primary Puppet server, and to compile configurations for Puppet agents.

- [Puppet commands](#) on page 305

Puppet's command line interface (CLI) consists of a single `puppet` command with many subcommands.

- [Running Puppet commands on Windows](#) on page 307

Puppet was originally designed to run on *nix systems, so its commands generally act the way *nix admins expect. Because Windows systems work differently, there are a few extra things to keep in mind when using Puppet commands.

- [Puppet agent on *nix systems](#) on page 314

Puppet agent is the application that manages the configurations on your nodes. It requires a Puppet primary server to fetch configuration catalogs from.

- [Puppet agent on Windows](#) on page 317

Puppet agent is the application that manages configurations on your nodes. It requires a Puppet primary server to fetch configuration catalogs.

- [Puppet apply](#) on page 320

Puppet apply is an application that compiles and manages configurations on nodes. It acts like a self-contained combination of the Puppet primary server and Puppet agent applications.

- [Puppet device](#) on page 322

With Puppet device, you can manage network devices, such as routers, switches, firewalls, and Internet of Things (IoT) devices, without installing a Puppet agent on them. Devices that cannot run Puppet applications require a Puppet agent to act as a proxy. The proxy manages certificates, collects facts, retrieves and applies catalogs, and stores reports on behalf of a device.

Puppet commands

Puppet's command line interface (CLI) consists of a single `puppet` command with many subcommands.

Puppet Server and Puppet's companion utilities [Facter](#) and [Hier](#), have their own CLI.

Puppet agent

Puppet agent is a core service that manages systems, with the help of a Puppet primary server. It requests a configuration catalog from a Puppet primary server, then ensures that all resources in that catalog are in their desired state.

For more information, see:

- [Overview of Puppet's architecture](#)
- Puppet [Agent on *nix systems](#)
- Puppet [Agent on Windows systems](#)

- Puppet [Agent's man page](#)

Puppet Server

Using Puppet code and various other data sources, Puppet Server compiles configurations for any number of Puppet agents.

Puppet Server is a core service and has its own subcommand, `puppetserver`, which isn't prefaced by the usual `puppet` subcommand.

For more information, see:

- [Overview of Puppet's architecture](#)
- [Puppet Server](#)
- [Puppet Server subcommands](#)
- Puppet [Primary Server's man page](#)

Puppet apply

Puppet apply is a core command that manages systems without contacting a Puppet primary server. Using Puppet modules and various other data sources, it compiles its own configuration catalog, and then immediately applies the catalog.

For more information, see:

- [Overview of Puppet's architecture](#)
- [Puppet apply](#)
- [Puppet apply's man page](#)

Puppet ssl

Puppet ssl is a command for managing SSL keys and certificates for Puppet SSL clients needing to communicate with your Puppet infrastructure.

Puppet ssl usage: `puppet ssl <action> [--certname <name>]`

Possible actions:

- `submit request`: Generate a certificate signing request (CSR) and submit it to the CA. If a private and public key pair already exist, they are used to generate the CSR. Otherwise, a new key pair is generated. If a CSR has already been submitted with the given `certname`, then the operation fails.
- `download_cert`: Download a certificate for this host. If the current private key matches the downloaded certificate, then the certificate is saved and used for subsequent requests. If there is already an existing certificate, it is overwritten.
- `verify`: Verify that the private key and certificate are present and match. Verify the certificate is issued by a trusted CA, and check the revocation status
- `bootstrap`: Perform all of the steps necessary to request and download a client certificate. If autosigning is disabled, then puppet will wait every `waitforcert` seconds for its certificate to be signed. To only attempt once and never wait, specify a time of 0. Since `waitforcert` is a Puppet setting, it can be specified as a time interval, such as 30s, 5m, 1h.

For more information, see the [SSL 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. Because Windows systems work differently, there are a few extra things to keep in mind when using Puppet commands.

Supported commands

Not all Puppet commands work on Windows. Notably, Windows nodes can't run the `puppet server` or `puppetserver ca` commands.

The following commands are designed for use on Windows:

- `puppet agent`
- `puppet apply`
- `puppet module`
- `puppet resource`
- `puppet config`
- `puppet lookup`
- `puppet help`

- [puppet man](#)

Running Puppet's commands

The installer adds Puppet commands to the PATH. After installing, you can run them from any command prompt (cmd.exe) or PowerShell prompt.

Open a new command prompt after installing. Any processes that were already running before you ran the installer do not pick up the changed PATH value.

Running with administrator privileges

You usually want to run Puppet's commands with administrator privileges.

Puppet has two privilege modes:

- Run with limited privileges, only manage certain resource types, and use a user-specific [confdir](#) and [codedir](#)
- Run with administrator privileges, manage the whole system, and use the system [confdir](#) and [codedir](#)

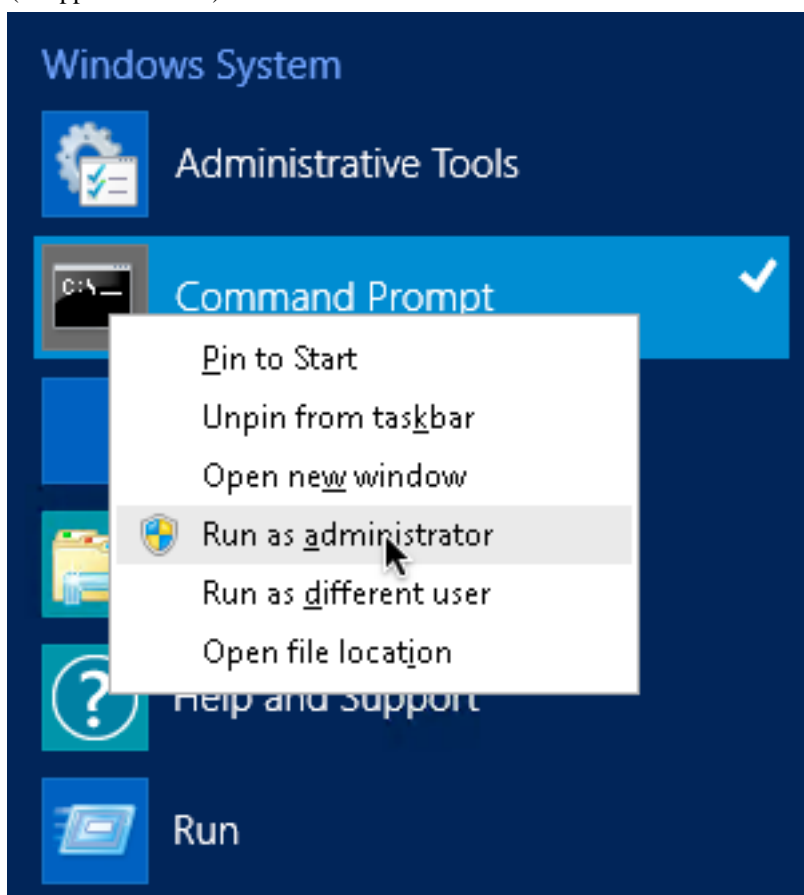
On *nix systems, Puppet defaults to running with limited privileges, when not run by `root`, but can have its privileges raised with the standard `sudo` command.

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

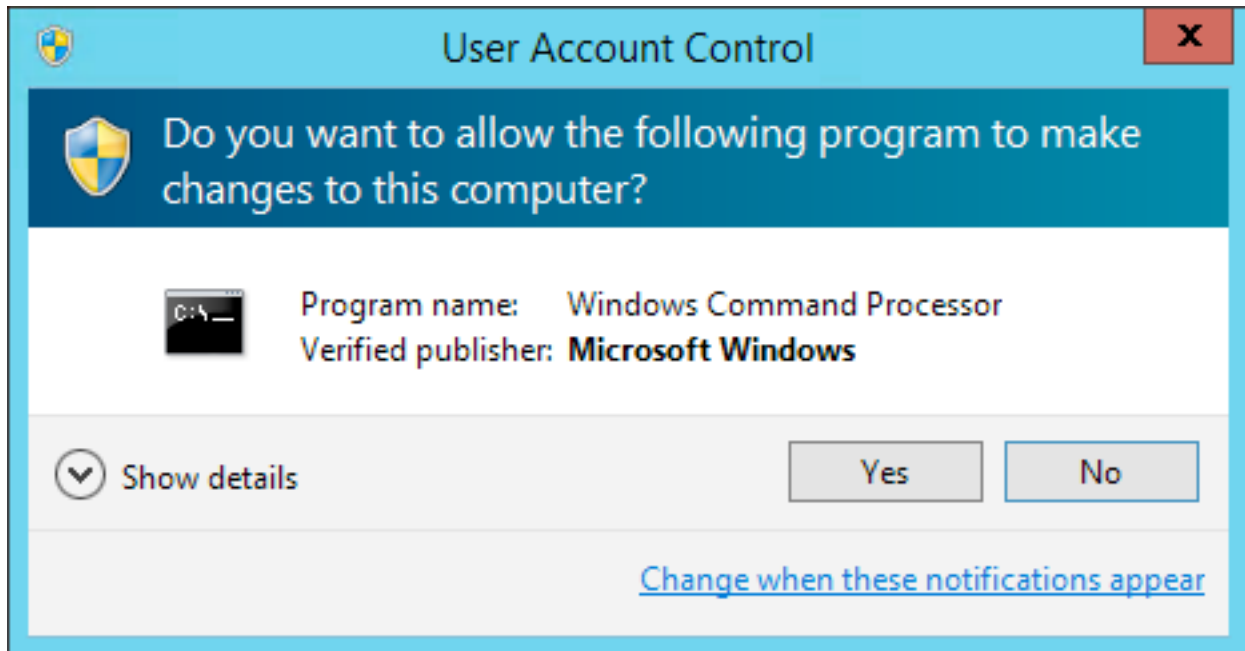
Newer versions of Windows manage security with User Account Control (UAC), which was added in Windows 2008 and Windows Vista. With UAC, most programs run by administrators still have limited privileges. To get administrator privileges, the process has to request those privileges when it starts.

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

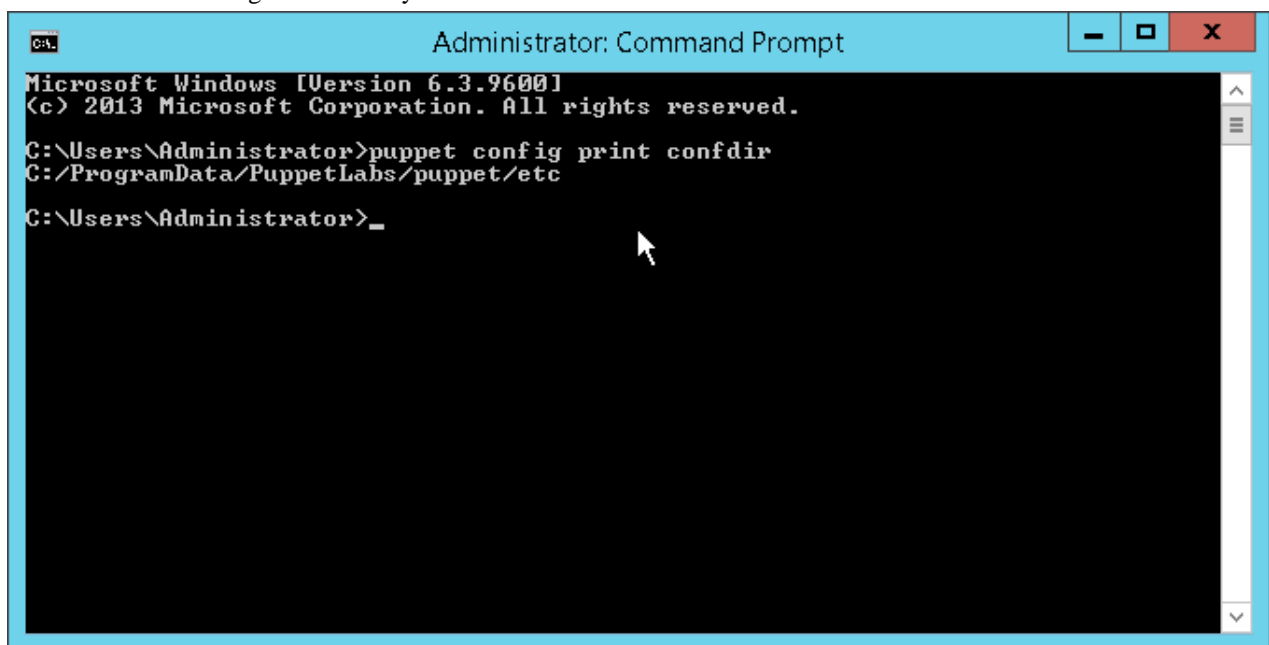
Right-click the **Start** (or apps screen tile) -> **Run as administrator**:



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

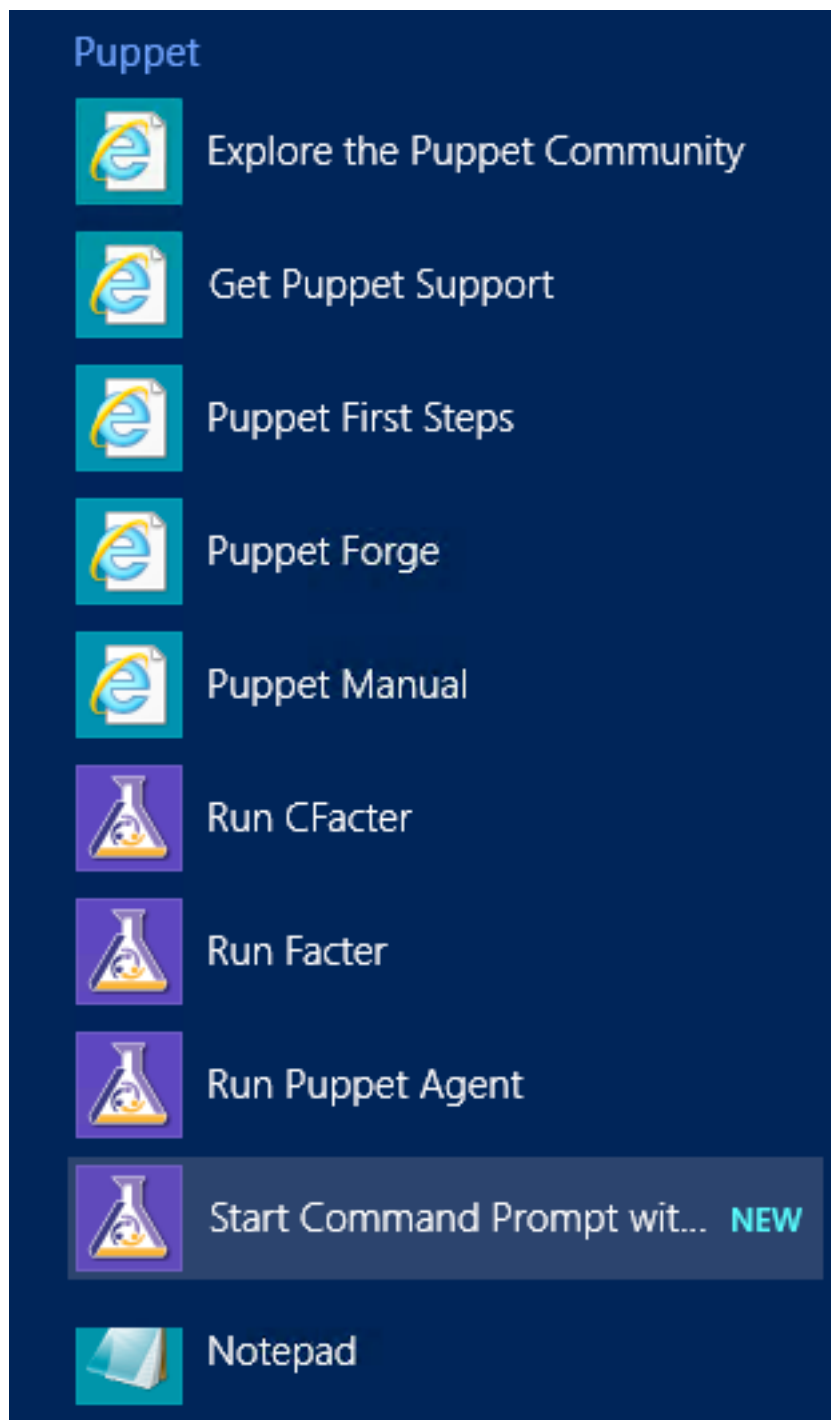


The title bar on the command prompt window begins with **Administrator**. This means Puppet commands that run from that window can manage the whole system.



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, because Puppet agent runs a normal [Windows service](#) and the Puppet commands work from any command or PowerShell prompt. They're provided solely as conveniences.

The **Start** menu items do the following:

Run Facter

This shortcut requests UAC elevation and, using the CLI, runs [Facter](#) with administrator privileges.

Run Puppet agent

This shortcut requests UAC elevation and, using the CLI, performs a single Puppet agent command with administrator privileges.

Start Command Prompt with Puppet

This shortcut starts a normal command prompt with the working directory set to Puppet's program directory. The CLI window icon is also set to the Puppet logo. This shortcut was particularly useful in previous versions of Puppet, before Puppet's commands were added to the PATH at installation time.

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.

primary Puppet server

About Puppet Server

Puppet is configured in an agent-server architecture, in which a primary server node manages the configuration information for a fleet of agent nodes. Puppet Server acts as the primary server node.

Puppet Server is a Ruby and Clojure application that runs on the Java Virtual Machine (JVM). Puppet Server runs Ruby code for compiling Puppet catalogs and for serving files in several JRuby interpreters. It also provides a certificate authority through Clojure.

This page describes the general requirements and the run environment for Puppet Server.

Supported Platforms

Puppet provides Puppet Server packages for the following platforms:

- Red Hat Enterprise Linux
- Centos
- Debian
- Ubuntu

If we do not provide a package for your system, you can run Puppet Server from source on any x86_64 Linux server with JDK 1.8 or 11. See [Running from source](#) on page 166 for more details.

Tip: Other POSIX compatible servers or versions of Java are unsupported by Puppet, but they may work. Join our [community Slack](#) for help with non-supported operating systems, architectures, or JRE versions.

Puppet Server releases

Puppet Server and Puppet share the same major release (Puppet Server 6.x and Puppet 6.x). However, they are versioned separately and might have different minor or patch versions (Puppet Server 6.5 versus Puppet 6.8). For a list of the maintained versions of Puppet, Puppet Server, and Puppet DB, see [Puppet releases and lifecycles](#).

Controlling the Service

The Puppet Server service name is `puppetserver`. To start and stop the service, use commands such as `service puppetserver restart`, `service puppetserver status` for your OS.

Puppet Server's Run Environment

Puppet Server consists of several related services. These services share state and route requests among themselves. The services run inside a single JVM process, using the Trapperkeeper service framework.

Embedded Web Server

Puppet Server uses a Jetty-based web server embedded in the service's JVM process. No additional or unique actions are required to configure and enable the web server.

You can modify the web server's settings in [webserver.conf](#) on page 82. You might need to edit this file if you use an external CA or run Puppet on a non-standard port.

Puppet API Service

Puppet Server provides APIs that are used by the Puppet agent to manage the configuration of your nodes. See [Puppet V3 HTTP API](#) for more information on the basic APIs.

Certificate Authority Service

Puppet Server includes a certificate authority (CA) service that:

- Accepts certificate signing requests (CSRs) from nodes.
- Serves certificates and a certificate revocation list (CRL) to nodes.
- Optionally accepts commands to sign or revoke certificates.

See [CA V1 HTTP API](#) for more information on these APIs.

Signing and revoking certificates over the network is disabled by default. You can use the `auth.conf` file to allow specific certificate owners the ability to issue commands.

The CA service stores credentials using `.pem` files. You can use the `puppetserver ca` command to interact with these credentials, including listing, signing, and revoking certificates.

Admin API Service

Puppet Server includes an administrative API for triggering maintenance tasks. The most common task refreshes Puppet's environment cache, which causes all of your Puppet code to reload without the requirement to restart the service. Consequently, you can deploy new code to long-timeout environments without executing a full restart of the service.

- For API docs, see:
 - [Environment cache](#) on page 202
 - [JRuby pool](#) on page 202
- For details about environment caching, see [About environments](#).

JRuby Interpreters

Most of Puppet Server's work is done by Ruby code running in JRuby. JRuby is an implementation of the Ruby interpreter that runs on the JVM.

You cannot use the `system gem` command to install Ruby Gems for the Puppet primary server. Instead, Puppet Server includes a separate `puppetserver gem` command for installing any libraries your Puppet extensions might require. See [Using Ruby gems](#) on page 138 for details.

If you want to test or debug code to be used by the Puppet Server, you can use the `puppetserver ruby` and `puppetserver irb` commands to execute Ruby code in a JRuby environment.

To handle parallel requests from agent nodes, Puppet Server maintains separate JRuby interpreters. These JRuby interpreters individually run Puppet's application code, and distribute agent requests among them.

You can configure the JRuby interpreters in the `jruby-puppet` section of [puppetserver.conf](#) on page 75.

Tuning Guide

You can maximize Puppet Server's performance by tuning your JRuby configuration. To learn more, see the Puppet Server [Tuning guide](#) on page 154.

User

Puppet Server needs to run as the user:

- `pe-puppet` if you are running Puppet Enterprise.
- `puppet` if you are running open source Puppet.

The user is specified in:

- `/etc/sysconfig/pe-puppetserver` for Puppet Enterprise.
- `/etc/sysconfig/puppetserver` for open source Puppet.

Note: Puppet Server ignores the `user` and `group` settings from `puppet.conf`.

All of the Puppet Server's files and directories must be readable and writable by this user.

Ports

By default, Puppet's HTTPS traffic uses port 8140. The OS and firewall must allow Puppet Server's JVM process to accept incoming connections on port 8140.

You can change the port in `webserver.conf` if necessary. See the [webserver.conf](#) on page 82 for details.

Logging

All of Puppet Server's logging is routed through the JVM [Logback](#) library. By default, it logs to `/var/log/puppetlabs/puppetserver/puppetserver.log`. The default log level is 'INFO'. By default, Puppet Server sends nothing to syslog.

All log messages follow the same path, including HTTP traffic, catalog compilation, certificate processing, and all other parts of Puppet Server's work.

Puppet Server also relies on Logback to manage, rotate, and archive Server log files. Logback archives Server logs when they exceed 200MB. Also, when the total size of all Server logs exceeds 1GB, Logback automatically deletes the oldest logs.

Logback is heavily configurable. If you need something more specialized than a unified log file, it may be possible to obtain. See [Logging](#) on page 74 for more details.

Finally, any errors that happen before logging is set up or cause the logging system to die, displays in `journalctl`.

SSL Termination

By default, Puppet Server handles SSL termination automatically.

For network configurations that require external SSL termination (e.g. with a hardware load balancer), additional configuration is required. See the [External SSL termination](#) on page 142 for details. In summary, you must:

- Configure Puppet Server to use HTTP instead of HTTPS.
- Configure Puppet Server to accept SSL information via insecure HTTP headers.
- Secure your network so that Puppet Server **cannot** be directly reached by **any** untrusted clients.

- Configure your SSL terminating proxy to set the following HTTP headers:
 - `X-Client-Verify` (mandatory).
 - `X-Client-DN` (mandatory for client-verified requests).
 - `X-Client-Cert` (optional; required for [trusted facts](#)).

Configuring Puppet Server

Puppet Server uses a combination of Puppet's configuration files along with its own configuration files, which are located in the `conf.d` directory. Refer to the [Config directory](#) for a list of Puppet's configuration files.

Puppet Server's `conf.d` directory contains:

- `global.conf`: Global configuration settings for Puppet Server.
- `webserver.conf` and `web-routes.conf`: Web server configuration settings.
- `puppetserver.conf`: Settings for Puppet Server itself, including the JRuby interpreter and the administrative API.
- `auth.conf`: Authentication rules for Puppet Server endpoints.
- `ca.conf`: Settings for the Certificate Authority service.

For detailed information about Puppet Server settings and the `conf.d` directory, refer to the [Configuring Puppet Server](#) on page 72 page.

Puppet Server uses Puppet's config files, including most of the settings in `puppet.conf`. However, Puppet Server treats some `puppet.conf` settings differently. You must be aware of these differences. You can view a complete list of these differences on the [Differing behavior in puppet.conf](#) on page 93 page.

Puppet agent on *nix systems

Puppet agent is the application that manages the configurations on your nodes. It requires a Puppet primary server 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 [serverport](#) setting on all agent nodes, and ensure that you change your primary Puppet server'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 <code>start</code> , <code>stop</code> , and <code>status</code> attributes to specify how non-root users can control the service.
ssh_authorized_key	
ssh_key	

Manage systems with Puppet agent

In a standard Puppet configuration, each node periodically does configuration runs to revert unwanted changes and to pick up recent updates.

On *nix nodes, there are three main ways to do this:

Run Puppet agent as a service.

The easiest method. The Puppet agent daemon does configuration runs at a set interval, which can be configured.

Make a cron job that runs Puppet agent.

Requires more manual configuration, but a good choice if you want to reduce the number of persistent processes on your systems.

Only run Puppet agent on demand.

You can also deploy [MCollective](#) to run on demand on many nodes.

Choose whichever one works best for your infrastructure and culture.

Run Puppet agent as a service

The `puppet agent` command can start a long-lived daemon process that does configuration runs at a set interval.

Note: If you are running Puppet agent as a non-root user, use a cron job instead.

1. Start the service.

The best method is with Puppet agent's init script / service configuration. When you install Puppet with packages, included is an init script or service configuration for controlling Puppet agent, usually with the service name `puppet` (for both open source and Puppet Enterprise).

In open source Puppet, enable the service by running this command:

```
sudo puppet resource service puppet ensure=running enable=true
```

You can also run the `sudo puppet agent` command with no additional options which causes the Puppet agent to start running and daemonize, however you won't have an interface for restarting or stopping it. To stop the daemon, use the process ID from the agent's `pidfile`:

```
sudo kill $(puppet config print pidfile --section agent)
```

2. (Optional) Configure the run interval.

The Puppet agent service defaults to doing a configuration run every 30 minutes. You can configure this with the `runinterval` setting in `puppet.conf` :

```
# /etc/puppetlabs/puppet/puppet.conf
[agent]
  runinterval = 2h
```

If you don't need frequent configuration runs, a longer run interval lets your primary Puppet server 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 primary Puppet server from getting overwhelmed, because the system time is probably synchronized across all of your agent nodes.

To set up a cron job, run the `puppet resource` command:

```
sudo puppet resource cron puppet-agent ensure=present user=root minute=30
  command='/opt/puppetlabs/bin/puppet agent --onetime --no-daemonize --splay
  --splaylimit 60'
```

The above example runs Puppet one time every hour.

Run Puppet agent on demand

Some sites prefer to run Puppet agent on-demand, and others use scheduled runs along with the occasional on-demand run.

You can start Puppet agent runs while logged in to the target system, or remotely with Bolt or MCollective.

Run Puppet agent on one machine, using SSH to log into it:

```
ssh ops@magpie.example.com sudo puppet agent --test
```

To run remotely on multiple machines, you need some form of orchestration or parallel execution tool, such as [Bolt](#) or [MCollective](#) with the [puppet agent plugin](#).

Note: As of Puppet agent 5.5.4, MCollective is deprecated and will be removed in a future version of Puppet agent. If you use Puppet Enterprise, consider migrating from [MCollective to Puppetorchestrator](#). If you use open source Puppet, migrate MCollective agents and filters using tools like [Bolt](#) and PuppetDB's [Puppet Query Language](#).

Disable and re-enable Puppet runs

Whether you're troubleshooting errors, working in a maintenance window, or developing in a sandbox environment, you might need to temporarily disable the Puppet agent from running.

1. To disable the agent, run:

```
sudo puppet agent --disable "<MESSAGE>"
```

2. To enable the agent, run:

```
sudo puppet agent --enable
```

Configuring Puppet agent

The Puppet agent comes with a default configuration that you might want to change.

Configure Puppet agent with [puppet.conf](#) using the [agent] section, the [main] section, or both. For information on settings relevant to Puppet agent, see [important settings](#).

Logging for Puppet agent on *nix systems

When running as a service, Puppet agent logs messages to syslog. Your syslog configuration determines where these messages are saved, but the default location is `/var/log/messages` on Linux, and `/var/log/system.log` on Mac OS X.

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 primary Puppet server 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 primary server to fetch configuration catalogs.

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 `serverport` setting on all agent nodes, and ensure that you change your Puppet primary server's port as well.

User

Puppet agent runs as the `LocalSystem` user, which lets it manage the configuration of the entire system, but prevents it from accessing files on UNC shares.

Puppet agent can also run as a different user. You can change the user in the Service Control Manager (SCM). To start the SCM, click **Start -> Run...** and then enter `Services.msc`.

You can also specify a different user when installing Puppet. To do this, install using the CLI and specify the required MSI properties: `PUPPET_AGENT_ACCOUNT_USER`, `PUPPET_AGENT_ACCOUNT_PASSWORD`, and `PUPPET_AGENT_ACCOUNT_DOMAIN`.

Puppet agent's user can be a local or domain user. If this user isn't already a local administrator, the Puppet installer adds it to the `Administrators` group. The installer also grants [Logon as Service](#) to the user.

Managing systems with Puppet agent

In a normal Puppet configuration, every node periodically does configuration runs to revert unwanted changes and to pick up recent updates.

On Windows nodes, there are two main ways to do this:

Run Puppet as a service.

The easiest method. The Puppet agent service does configuration runs at a set interval, which can be configured.

Run Puppet agent on demand.

You can also use [Bolt](#) or deploy [MCollective](#) to run on demand on many nodes.

Because the Windows version of the Puppet agent service is much simpler than the *nix version, there's no real performance to be gained by running Puppet as a scheduled task. If you want scheduled configuration runs, use the Windows service.

Running Puppet agent as a service

The Puppet installer configures Puppet agent to run as a Windows service and starts it. No further action is needed. Puppet agent does configuration runs at a set interval.

Configuring the run interval

The Puppet agent service defaults to doing a configuration run every 30 minutes. If you don't need frequent configuration runs, a longer run interval lets your Puppet primary servers handle many more agent nodes.

You can configure this with the [runinterval](#) setting in `puppet.conf`:

```
# C:\ProgramData\PuppetLabs\puppet\etc\puppet.conf
[agent]
  runinterval = 2h
```

After you change the run interval, the next run happens on the previous schedule, and subsequent runs happen on the new schedule.

Configuring the service start up type

The Puppet agent service defaults to starting automatically. If you want to start it manually or disable it, you can configure this during installation.

To do this, install using the CLI and specify the [PUPPET_AGENT_STARTUP_MODE](#) MSI property.

You can also configure this after installation with the Service Control Manager (SCM). To start the SCM, click **Start** -> **Run...** and enter `Services.msc`.

You can also configure agent service with the `sc.exe` command. To prevent the service from starting on boot, run the following command from the Command Prompt (`cmd.exe`):

```
sc config puppet start= demand
```

Important: The space after `start=` is mandatory and must be run in `cmd.exe`. This command won't work from PowerShell.

To stop and restart the service, run the following commands:

```
sc stop puppet
sc start puppet
```

To change the arguments used when triggering a Puppet agent run, add flags to the command:

```
sc start puppet --debug --logdest eventlog
```

This example changes the level of detail that gets written to the Event Log.

Running Puppet agent on demand

Some sites prefer to run Puppet agent on demand, and others occasionally need to do an on-demand run.

You can start Puppet agent runs while logged in to the target system, or remotely with [Bolt](#) or [MCollective](#).

While logged in to the target system

On Windows, log in as an administrator, and start the configuration run by selecting **Start** -> **Run Puppet Agent**. If Windows prompts for User Account Control confirmation, click **Yes**. The status result of the run is shown in a command prompt window.

Running other Puppet commands

To run other Puppet-related commands, start a command prompt with administrative privileges. You can do so by right-clicking the **Command Prompt** or **Start Command Prompts with Puppet** program and clicking **Run as administrator**. Click Yes if the system asks for UAC confirmation.

Remotely

Open source Puppet users can use [Bolt](#) to run tasks and commands on remote systems.

Alternatively, you can install MCollective and the [puppet agent plugin](#) to get similar capabilities, but Puppet doesn't provide standalone MCollective packages for Windows.

Important: As of Puppet agent 5.5.4, MCollective is deprecated and will be removed in a future version of Puppet agent. If you use Puppet Enterprise, consider migrating from [MCollective to Puppet orchestrator](#). If you use open source Puppet, migrate MCollective agents and filters using tools like [Bolt](#) and the [PuppetDB Puppet Query Language](#).

Disabling and re-enabling Puppet runs

Whether you're troubleshooting errors, working in a maintenance window, or developing in a sandbox environment, you might need to temporarily disable the Puppet agent from running.

1. Start a command prompt with **Run as administrator**.
2. To disable the agent, run:

```
puppet agent --disable "<MESSAGE>"
```

3. To enable the agent, run:

```
puppet agent --enable
```

Configuring Puppet agent on Windows

The Puppet agent comes with a default configuration that you might want to change.

Configure Puppet agent with [puppet.conf](#), using the [agent] section, the [main] section, or both. For more information on which settings are relevant to Puppet agent, see [important settings](#).

Logging for Puppet agent on Windows systems

When running as a service, Puppet agent logs messages to the Windows Event Log. You can view its logs by browsing the **Event Viewer**. Click **Control Panel** -> **System and Security** -> **Administrative Tools** -> **Event Viewer**.

By default, Puppet logs to the Application event log. However, you can configure Puppet to log to a separate Puppet log instead.

To enable the Puppet log, create the requisite registry key by opening a command prompt and running one of the following commands:

Bash:

```
reg add HKLM\System\CurrentControlSet\Services\EventLog\Puppet\Puppet /v
  ErrorMessageFile /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, 6.4.4 on the 6.4.x stream and 6.8.0 on the primary server 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 are not moved; only new messages are recorded in the newly created Puppet log.

You can adjust how verbose the logs are with the `log_level` setting, which defaults to `notice`.

When running in the foreground with the `--verbose`, `--debug`, or `--test` options, Puppet agent logs directly to the terminal.

When started with the `--logdest <FILE>` option, Puppet agent logs to the file specified by `<FILE>`. Note that there are no file size checks for the `--logdest <FILE>` option.

Reporting for Puppet agent on Windows systems

In addition to local logging, Puppet agent submits a report to the primary server 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 primary server 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 primary server and Puppet agent applications.

For details about invoking the `puppet apply` command, see the [puppet apply man page](#).

Supported platforms

Puppet apply runs similarly on *nix and Windows systems. Not all operating systems can manage the same resources with Puppet; some resource types are OS-specific, and others have OS-specific features. For more information, see the [resource type reference](#).

Puppet apply's run environment

Unlike Puppet agent, Puppet apply never runs as a daemon or service. It runs as a single task in the foreground, which compiles a catalog, applies it, files a report, and exits.

By default, it never initiates outbound network connections, although it can be configured to do so, and it never accepts inbound network connections.

Main manifest

Like the primary Puppet server application, Puppet apply uses its [settings](#) (such as `basemodulepath`) and the configured [environments](#) to locate the Puppet code and configuration data it uses when compiling a catalog.

The one exception is the [main manifest](#). Puppet apply always requires a single command line argument, which acts as its main manifest. It ignores the main manifest from its environment.

Alternatively, you can write a main manifest directly using the command line, with the `-e` option. For more information, see the [puppet apply man page](#).

User

Puppet apply runs as whichever user executed the Puppet apply command.

To manage a complete system, 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
<code>augeas</code>	
<code>cron</code>	Only non-root cron jobs can be viewed or set.
<code>exec</code>	Cannot run as another user or group.
<code>file</code>	Only if the non-root user has read/write privileges.
<code>notify</code>	
<code>schedule</code>	
<code>service</code>	For services that don't require root. You can also use the <code>start</code> , <code>stop</code> , and <code>status</code> attributes to specify how non-root users can control the service. For more information, see tips and examples for the service type.
<code>ssh_authorized_key</code>	
<code>ssh_key</code>	

To install packages into a directory controlled by a non-root user, you can either use an `exec` to unzip a tarball or use a recursive `file` resource to copy a directory into place.

Network access

By default, Puppet apply does not communicate over the network. It uses its local collection of modules for any file sources, and does not submit reports to a central server.

Depending on your system and the resources you are managing, it might download packages from your configured package repositories or access files on UNC shares.

If you have configured an [external node classifier \(ENC\)](#), your ENC script might create an outbound HTTP connection. Additionally, if you've configured the [HTTP report processor](#), Puppet agent sends reports via HTTP or HTTPS.

If you have configured PuppetDB, Puppet apply creates outbound HTTPS connections to PuppetDB.

Logging

Puppet apply logs directly to the terminal, which is good for interactive use, but less so when running as a scheduled task or cron job.

You can adjust how verbose the logs are with the `log_level` setting, which defaults to `notice`. Setting it to `info` is equivalent to running with the `--verbose` option, and setting it to `debug` is equivalent to `--debug`. You can also make logs quieter by setting it to `warning` or lower.

When started with the `--logdest syslog` option, Puppet apply logs to the *nix syslog service. Your syslog configuration dictates where these messages are saved, but the default location is `/var/log/messages` on Linux, and `/var/log/system.log` on Mac OS X.

When started with the `--logdest eventlog` option, it logs to the Windows Event Log. You can view its logs by browsing the **Event Viewer**. Click **Control Panel -> System and Security -> Administrative Tools -> Event Viewer**.

When started with the `--logdest <FILE>` option, it logs to the file specified by `<FILE>`.

Reporting

In addition to local logging, Puppet apply processes a report using its configured [report handlers](#), like a primary Puppet server does. Using the `reports` setting, you can enable different reports. For more information, see the list of available [reports](#). For information about reporting, see the [reporting](#) documentation.

To disable reporting and avoid taking up disk space with the `store` report handler, you can set `report = false` in `puppet.conf`.

Managing systems with Puppet apply

In a typical site, every node periodically does a Puppet run, to revert unwanted changes and to pick up recent updates.

Puppet apply doesn't run as a service, so you must manually create a scheduled task or cron job if you want it to run on a regular basis, instead of using Puppet agent.

On *nix, you can use the `puppet resource` command to set up a cron job.

This example runs Puppet one time per hour, with Puppet Enterprise paths:

```
sudo puppet resource cron puppet-apply ensure=present user=root minute=60
  command='/opt/puppetlabs/bin/puppet apply /etc/puppetlabs/puppet/manifests
  --logdest syslog'
```

Configuring Puppet apply

Configure Puppet apply in the `puppet.conf` file, using the `[user]` section, the `[main]` section, or both.

For information on which settings are relevant to `puppet apply`, see [important settings](#).

Puppet device

With Puppet device, you can manage network devices, such as routers, switches, firewalls, and Internet of Things (IOT) devices, without installing a Puppet agent on them. Devices that cannot run Puppet applications require a Puppet agent to act as a proxy. The proxy manages certificates, collects facts, retrieves and applies catalogs, and stores reports on behalf of a device.

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

Note: If you are writing a module for a remote resource, we recommend using transports instead of devices. Transports have extended functionality and can be used with other workflows, such as with [Bolt](#). For more information on transports and how to port your existing code, see [Resource API Transports](#).

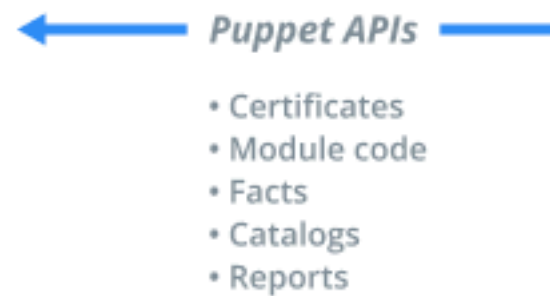
The Puppet device model

In a typical deployment model, a Puppet agent is installed on each system managed by Puppet. However, not all systems can have agents installed on them.

For these devices, you can configure a Puppet agent on another system which connects to the API or CLI of the device, and acts as a proxy between the device and the primary Puppet server.

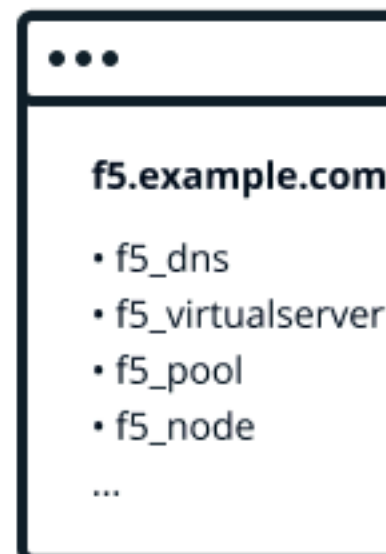
In the diagram below, Puppet device is on a proxy Puppet agent (agent.example.com) and is being used to manage an F5 load balancer (f5.example.com) and a Cisco switch (cisco.example.com).

Primary Server



F5 REST

F5 device



Puppet device's run environment

Puppet device runs as a single process in the foreground that manages devices, rather than as a daemon or service like a Puppet agent.

User

The `puppet device` command runs with the privileges of the user who runs it.

Run Puppet device as:

- Root on *nix
- Either LocalService or a member of the Administrators group on Windows

Logging

By default, Puppet device outputs directly to the terminal, which is valuable for interactive use. When you run it as a cron job or scheduled task, use the `logdest` option to direct the output to a file.

On *nix, run Puppet device with the `--logdest syslog` option to log to the *nix syslog service:

```
puppet device --verbose --logdest syslog
```

Your syslog configuration determines where these messages are saved, but the default location is `/var/log/` messages on Linux, and `/var/log/system.log` on Mac OS X. 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 primary Puppet server 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 primary Puppet server.

For example, to install the [f5](#) and [cisco_ios](#) device modules on the primary server, run the following commands:

```
$ sudo puppet module install f5-f5
```

```
$ sudo puppet module install puppetlabs-cisco_ios
```

Configuring Puppet device on the proxy Puppet agent

You can specify multiple devices in `device.conf`, which is configurable with the `deviceconfig` setting on the proxy agent.

For example, to configure an F5 and a Cisco IOS device, add the following lines to the `device.conf` file:

```
[f5.example.com]
type f5
url https://username:password@f5.example.com

[cisco.example.com]
type cisco_ios
url file:///etc/puppetlabs/puppet/devices/cisco.example.com.yaml
```

The string in the square brackets is the device's certificate name — usually the hostname or FQDN. The certificate name is how Puppet identifies the device.

For the `url`, specify the device's connection string. The connection string varies by device module. In the first example above, the F5 device connection credentials are included in the `url` `device.conf` file, because that is how the F5 module stores credentials. However, the Cisco IOS module uses the Puppet Resource API, which stores that information in a separate credentials file. So, Cisco IOS devices would also have a `/etc/puppetlabs/puppet/devices/<device cert name>.conf` file similar to the following content:

```
{
  "address": "cisco.example.com"
  "port": 22
  "username": "username"
  "password": "password"
  "enable_password": "password"
}
```

For more information, see [device.conf](#).

Classify the proxy Puppet agent for the device

Some device modules require the proxy Puppet agent to be classified with the base class of the device module to install or configure resources required by the module. Refer to the specific device module README for details.

To classify proxy Puppet agent:

1. Classify the agent with the base class of the device module, for each device it manages in the manifest. For example:

```
node 'agent.example.com' {
  include cisco_ios
  include f5
}
```

2. Apply the classification by running `puppet agent -t` on the proxy Puppet agent.

Classify the device

Classify the device with resources to manage its configuration.

The examples below manage DNS settings on an F5 and a Cisco IOS device.

1. In the `site.pp` manifest, declare DNS resources for the devices. For example:

```
node 'f5.example.com' {
  f5_dns { '/Common/dns':
    name_servers => ['4.2.2.2.', '8.8.8.8'],
    same        => ['localhost', 'example.com'],
  }
}

node 'cisco.example.com' {
  network_dns { 'default':
    servers => [4.2.2.2, '8.8.8.8'],
    search => ['localhost', 'example.com'],
  }
}
```

2. Apply the manifest by running `puppet device -v` on the proxy Puppet agent.

Note: Resources vary by device module. Refer to the specific device module README for details.

Get and set data using Puppet device

The traditional Puppet `apply` and Puppet resource applications cannot target device resources: running `puppet resource --target <DEVICE>` does not return data from the target device. Instead, use Puppet device to get data from devices, and to set data on devices. The following are optional parameters.

Get device data with the `resource` parameter

Syntax:

```
puppet device --resource <RESOURCE> --target <DEVICE>
```

Use the `resource` parameter to retrieve resources from the target device. For example, to return the DNS values for example F5 and Cisco IOS devices:

```
sudo puppet device --resource f5_dns --target f5.example.com
sudo puppet device --resource network_dns --target cisco.example.com
```

Set device data with the `apply` parameter

Syntax:

```
puppet device --verbose --apply <FILE> --target <DEVICE>
```

Use the `--apply` parameter to set a local manifest to manage resources on a remote device. For example, to apply a Puppet manifest to the F5 and Cisco devices:

```
sudo puppet device --verbose --apply manifest.pp --target f5.example.com
sudo puppet device --verbose --apply manifest.pp --target cisco.example.com
```

View device facts with the `facts` parameter

Syntax:

```
puppet device --verbose --facts --target <DEVICE>
```

Use the `--facts` parameter to display the facts of a remote target. For example, to display facts on a device:

```
sudo puppet device --verbose --facts --target f5.example.com
```

Managing devices using Puppet device

Running the `puppet device` or `puppet-device` command (without `--resource` or `--apply` options) tells the proxy agent to retrieve catalogs from the primary server and apply them to the remote devices listed in the `device.conf` file.

To run Puppet device on demand and for all of the devices in `device.conf`, run:

```
sudo puppet device --verbose
```

To run Puppet device for only one of the multiple devices in the `device.conf` file, specify a `--target` option:

```
$ sudo puppet device -verbose --target f5.example.com
```

To run Puppet device on a specific group of devices, as opposed to all devices in the `device.conf` file, create a separate configuration file containing the devices you want to manage, and specify the file with the `--deviceconfig` option:

```
$ sudo puppet device --verbose --deviceconfig /path/to/custom-device.conf
```

To set up a cron job to run Puppet device on a recurring schedule, run:

```
$ sudo puppet resource cron puppet-device ensure=present user=root minute=30
  command='/opt/puppetlabs/bin/puppet device --verbose --logdest syslog'
```

Example

Follow the steps below to run Puppet device in a production environment, using `cisco_ios` as an example.

1. Install the module on the primary Puppet server: `sudo puppet module install puppetlabs-cisco_ios`.
2. Include the module on the proxy Puppet agent by adding the following line to the primary server'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 primary server: `sudo puppetserver ca 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.

<code>--debug</code> or <code>-d</code>	Enables debugging
<code>--trace</code> or <code>-t</code>	Enables stack tracing if Ruby fails
<code>--verbose</code> or <code>-v</code>	Enables detailed reporting

Classifying nodes

You can classify nodes using an external node classifier (ENC), which is a script or application that tells Puppet which classes a node must have. It can replace or work in concert with the node definitions in the main site manifest (`site.pp`).

The `external_nodes` script receives the name of the node to classify as its first argument, which is usually the node's fully qualified domain name. For more information, see the [configuration reference](#).

Depending on the external data sources you use in your infrastructure, building an external node classifier can be a valuable way to extend Puppet.

Note: You can use an ENC instead of or in combination with [node definitions](#).

External node classifiers

An external node classifier is an executable that Puppet Server or `puppet apply` can call; it doesn't have to be written in Ruby. Its only argument is the name of the node to be classified, and it returns a YAML document describing the node.

Inside the ENC, you can reference any data source you want, including [PuppetDB](#). From Puppet's perspective, the ENC submits a node name and gets back a hash of information.

External node classifiers can co-exist with standard node definitions in `site.pp`; the classes declared in each source are merged together.

Merging classes from multiple sources

Every node always gets a node object from the configured node terminus. The node object might be empty, or it might contain classes, parameters, and an environment. The [node terminus setting](#), `node_terminus`, takes effect where the catalog is compiled, on Puppet Server when using an agent-server configuration, and on the node itself when using `puppet apply`. The default node terminus is `plain`, which returns an empty node object, leaving node configuration to the main manifest. The `exec` terminus calls an ENC script to determine what goes in the node object. Every node might also get a [node definition](#) from the [main manifest](#).

When compiling a node's catalog, Puppet includes all of the following:

- Classes specified in the node object it received from the node terminus.
- Classes or resources that are in the site manifest but outside any node definitions.

- Classes or resources in the most specific node definition in `site.pp` that matches the current node (if `site.pp` contains any node definitions). The following notes apply:
 - If `site.pp` contains at least one node definition, it must have a node definition that matches the current node; compilation fails if a match can't be found.
 - If the node name resembles a dot-separated fully qualified domain name, Puppet makes multiple attempts to match a node definition, removing the right-most part of the name each time. Thus, Puppet would first try `agent1.example.com`, then `agent1.example`, then `agent1`. This behavior isn't mimicked when calling an ENC, which is invoked only once with the agent's full node name.
 - If no matching node definition can be found with the node's name, Puppet tries one last time with a node name of `default`; most users include a `node default { }` statement in their `site.pp` file. This behavior isn't mimicked when calling an ENC.

Comparing ENCs and node definitions

If you're trying to decide whether to use an ENC or main manifest node definitions (or both), consider the following:

- The YAML returned by an ENC isn't an exact equivalent of a node definition in `site.pp` — it can't declare individual resources, declare relationships, or do conditional logic. An ENC can only declare classes, assign top-scope variables, and set an environment. So, an ENC is most effective if you've done a good job of separating your configurations out into classes and modules.
- ENCs can set an environment for a node, overriding whatever environment the node requested.
- Even if you aren't using node definitions, you can still use `site.pp` to do things like set global resource defaults.
- Unlike regular node definitions, where a node can match a less specific definition if an exactly matching definition isn't found (depending on Puppet's `strict_hostname_checking` setting), an ENC is called only once, with the node's full name.

Connect an ENC

Configure two settings to have Puppet Server connect to an external node classifier.

In the primary server's `puppet.conf` file:

1. Set the `node_terminus` setting to `exec`.
2. Set the `external_nodes` setting to the path to the ENC executable.

For example:

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

ENC output format

An ENC must return either nothing or a YAML hash to standard out. The hash must contain at least one of `classes` or `parameters`, or it can contain both. It can also optionally contain an `environment` key.

ENCs exit with an exit code of 0 when functioning normally, and can exit with a non-zero exit code if you want Puppet to behave as though the requested node was not found.

If an ENC returns nothing or exits with a non-zero exit code, the catalog compilation fails with a “could not find node” error, and the node is unable to retrieve configurations.

For information about the YAML format, see yaml.org.

Classes

If present, the value of `classes` must be either an array of class names or a hash whose keys are class names. That is, the following are equivalent:

```
classes:
  - common
  - puppet
  - dns
  - ntp

classes:
  common:
  puppet:
  dns:
  ntp:
```

If you're specifying parameterized classes, use the hash key syntax, not the array syntax. The value for a parameterized class is a hash of the class's parameters and values. Each value can be a string, number, array, or hash. Put string values in quotation marks, because YAML parsers sometimes treat certain unquoted strings (such as `on`) as Booleans. Non-parameterized classes can have empty values.

```
classes:
  common:
  puppet:
  ntp:
    ntpserver: 0.pool.ntp.org
  aptsetup:
    additional_apt_repos:
      - deb localrepo.example.com/ubuntu lucid production
      - deb localrepo.example.com/ubuntu lucid vendor
```

Parameters

If present, the value of the `parameters` key must be a hash of valid variable names and associated values; these are exposed to the compiler as top-scope variables. Each value can be a string, number, array, or hash.

```
parameters:
  ntp_servers:
    - 0.pool.ntp.org
    - ntp.example.com
  mail_server: mail.example.com
  iburst: true
```

Environment

If present, the value of `environment` must be a string representing the desired [environment](#) for this node. This is the only environment used by the node in its requests for catalogs and files.

```
environment: production
```

Complete example

```
---
classes:
  common:
  puppet:
  ntp:
    ntpserver: 0.pool.ntp.org
```

```

aptsetup:
  additional_apt_repos:
    - deb localrepo.example.com/ubuntu lucid production
    - deb localrepo.example.com/ubuntu lucid vendor
parameters:
  ntp_servers:
    - 0.pool.ntp.org
    - ntp.example.com
  mail_server: mail.example.com
  iburst: true
environment: production

```

Puppet reports

Puppet creates a report about its actions and your infrastructure each time it applies a catalog during a Puppet run. You can create and use report processors to generate insightful information or alerts from those reports.

- [Reporting](#) on page 332

In a client-server configuration, an agent sends its report to the primary server for processing. In a standalone configuration, the `puppet apply` command processes the node's own reports. In both configurations, report processor plugins handle received reports. If you enable multiple report processors, Puppet runs all of them for each report.

- [Report reference](#) on page 333

Puppet has a set of built-in report processors, which you can configure.

- [Writing custom report processors](#) on page 334

Create and use report processors to generate insightful information or alerts from Puppet reports. You can write your own report processor in Ruby and include it in a Puppet module. Puppet uses the processor to send report data to the service in the format you defined.

- [Report format](#) on page 335

Puppet versions 7 and later generate report format 12. This format is backward compatible with report formats 9-11.

Reporting

In a client-server configuration, an agent sends its report to the primary server for processing. In a standalone configuration, the `puppet apply` command processes the node's own reports. In both configurations, report processor plugins handle received reports. If you enable multiple report processors, Puppet runs all of them for each report.

Each processor typically does one of two things with the report:

- It sends some of the report data to another service, such as PuppetDB, that can collate it.
- It triggers alerts on another service if the data matches a specified condition, such as a failed run.

That external service can then provide a way to view the processed report.

Report data

Puppet report processors handle these points of data:

- Metadata about the node, its environment and Puppet version, and the catalog used in the run.
- The status of every resource.
- Actions, also called events, taken during the run.
- Log messages generated during the run.
- Metrics about the run, such as its duration and how many resources were in a given state.

That external service can then provide a way to view the processed report.

Configuring reporting

An agent sends reports to the primary server by default. You can turn off reporting by changing the `report` setting in an agent's `puppet.conf` file.

On primary servers and on nodes running Puppet apply, you can configure enabled report processors as a comma-separated list in the `reports` setting. The default `reports` value is `'store'`, which stores reports in the configured `reportdir`.

To turn off reports entirely, set `reports` to `'none'`.

For details about configuration settings in Puppet, see the [Configuration reference](#).

Accessing reports

There are multiple ways to access Puppet report data:

- In Puppet Enterprise (PE), view run logs and event reports on the **Reports** page. See [Infrastructure reports](#).
- In PuppetDB, with its [report processor enabled](#), interface with third-party tools such as [Puppetboard](#) or [PuppetExplorer](#).
- Use one of the [built-in report processors](#). For example, the `http` processor sends YAML dumps of reports through POST requests to a designated URL; the `log` processor saves received logs to a local `log` file.
- Use a report processor from a module, such as [tagmail](#).
- Query PuppetDB for stored report data and build your own tools to display it. For details about the types of data that PuppetDB collects and the API endpoints it uses, see the [API documentation](#) for the endpoints `events`, `event-counts`, and `aggregate-event-counts`.
- Write a custom report processor.

Related information

[puppet.conf: The main config file](#) on page 62

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

[Writing custom report processors](#) on page 334

Create and use report processors to generate insightful information or alerts from Puppet reports. You can write your own report processor in Ruby and include it in a Puppet module. Puppet uses the processor to send report data to the service in the format you defined.

[Report format](#) on page 335

Puppet versions 7 and later generate report format 12. This format is backward compatible with report formats 9-11.

Report reference

Puppet has a set of built-in report processors, which you can configure.

By default, after applying a catalog, Puppet generates a report that includes information about the run: events, log messages, resource statuses, metrics, and metadata. Each host sends its report as a YAML dump.

The agent sends its report to the primary server for processing, whereas agents running `puppet apply` process their own reports. Either way, Puppet handles every report with a set of report processors, which are specified in the `reports` setting in the agent's `puppet.conf` file.

By default, Puppet uses the `store` report processor. You can enable other report processors or disable reporting in the `reports` setting.

http

Sends reports via HTTP or HTTPS. This report processor submits reports as POST requests to the address in the `reporturl` setting. When you specify an HTTPS URL, the remote server must present a certificate issued

by the Puppet CA or the connection fails validation. The body of each POST request is the YAML dump of a `Puppet::Transaction::Report` object, and the content type is set as `application/x-yaml`.

log

Sends all received logs to the local log destinations. The usual log destination is `syslog`.

store

Stores the `yaml` report in the configured `reportdir`. By default, this is the report processor Puppet uses. These files collect quickly — one every half hour — so be sure to perform maintenance on them if you use this report.

Writing custom report processors

Create and use report processors to generate insightful information or alerts from Puppet reports. You can write your own report processor in Ruby and include it in a Puppet module. Puppet uses the processor to send report data to the service in the format you defined.

A report processor must follow these rules:

- The processor name must be a valid Ruby symbol that starts with a letter and contains only alphanumeric characters.
- The processor must be in its own Ruby file, `<PROCESSOR_NAME>.rb`, and stored inside the Puppet module directory `lib/puppet/reports/`
- The processor code must start with `require 'puppet'`
- The processor code must call the method `Puppet::Reports.register_report(:NAME)`. This method takes the name of the report as a symbol, and a mandatory block of code with no arguments that contains:
 - A Markdown-formatted string describing the processor, passed to the `desc(<DESCRIPTION>)` method.
 - An implementation of a method named `process` that contains the report processor's main functionality.

Puppet lets the `process` method access a `self` object, which will be a `Puppet::Transaction::Report` object describing a Puppet run.

The processor can access report data by calling accessor methods on `self`, and it can forward that data to any service you configure in the report processor. It can also call `self.to_yaml` to dump the entire report to YAML. Note that the YAML output isn't a safe, well-defined data format — it's a serialized object.

Example report processor

To use this report processor, include it in the comma-separated list of processors in the Puppet primary server's `reports` setting in `puppet.conf`: `reports = store,myreport`.

```
# Located in /etc/puppetlabs/puppet/modules/myreport/lib/puppet/reports/
myreport.rb.
require 'puppet'
# If necessary, require any other Ruby libraries for this report here.

Puppet::Reports.register_report(:myreport) do
  desc "Process reports via the fictional my_cool_cmdb API."

  # Declare and configure any settings here. We'll pretend this connects to
  our API.
  my_api = MY_COOL_CMD

  # Define and configure the report processor.
  def process
    # Do something that sets up the API we're sending the report to here.
    # For instance, let's check on the node's status using the report object
    (self):
    if self.status != nil then
      status = self.status
```

```

    else
      status = 'undefined'
    end

    # Next, let's do something if the status equals 'failed'.
    if status == 'failed' then
      # Finally, dump the report object to YAML and post it using the API
    object:
      my_api.post(self.to_yaml)
    end
  end
end

```

To use this report processor, include it in the comma-separated list of processors in the Puppet primary server's `reports` setting in `puppet.conf`:

```
reports = store,myreport
```

For more examples using this API, see [the built-in reports' source code](#) or one of these custom reports created by a member of the Puppet community:

- [Report failed runs to Jabber/XMPP](#)
- [Send metrics to a Ganglia server via gmetric](#)

These community reports aren't provided or supported by Puppet, Inc.

Related information

[Report format](#) on page 335

Puppet versions 7 and later generate report format 12. This format is backward compatible with report formats 9-11.

Report format

Puppet versions 7 and later generate report format 12. This format is backward compatible with report formats 9-11.

Puppet::Transaction::Report

Property	Type	Description
host	string	The host that generated this report.
time	datetime	When the Puppet run began.
logs	array	Zero or more Puppet::Util::Log objects.
metrics	hash	Maps from string (metric category) to Puppet::Util::Metric.
resource_statuses	hash	Maps from resource name to Puppet::Resource::Status
configuration_version	string or integer	The configuration version of the Puppet run. This is a string for user-specified versioning schemes. Otherwise it is an integer representing seconds since the Unix epoch.
transaction_uuid	string	A UUID covering the transaction. The query parameters for the catalog retrieval include the same UUID.

Property	Type	Description
code_id	string	The ID of the code input to the compiler.
job_id	string, or null	The ID of the job in which this transaction occurred.
catalog_uuid	string	A primary server generated catalog UUID, useful for connecting a single catalog to multiple reports.
server_used	string	The name of the primary server used to compile the catalog. If failover occurred, this holds the first primary server successfully contacted. If this run had no primary server (for example, a <code>puppet apply</code> run), this field is blank.
report_format	string or integer	"10" or 10
puppet_version	string	The version of the Puppet agent.
status	string	The transaction status: failed, changed, or unchanged.
transaction_completed	Boolean	Whether the transaction completed. For instance, if the transaction had an unrescued exception, <code>transaction_completed = false</code> .
noop	Boolean	Whether the Puppet run was in no-operation mode when it ran.
noop_pending	Boolean	Whether there are changes that were not applied because of no-operation mode.
environment	string	The environment that was used for the Puppet run.
corrective_change	Boolean	True if a change or no-operation event in this report was caused by an unexpected change to the system between Puppet runs.
cached_catalog_status	string	The status of the cached catalog used in the run: <code>not_used</code> , <code>explicitly_requested</code> , or <code>on_failure</code> .

Puppet::Util::Log

Property	Type	Description
file	string	The path and filename of the manifest file that triggered the log message. This property is not always present.

Property	Type	Description
line	integer	The manifest file's line number that triggered the log message. This property is not always present.
level	symbol	The severity level of the message :debug, :info, :notice, :warning, :err, :alert, :emerg, :crit.
message	string	The text of the message.
source	string	The origin of the log message. This could be a resource, a property of a resource, or the string "Puppet".
tags	array	Each array element is a string.
time	datetime	The time at which the message was sent.

Puppet::Util::Metric

A `Puppet::Util::Metric` object represents all the metrics in a single category.

Property	Type	Description
name	string	Specifies the name of the metric category. This is the same as the key associated with this metric in the metrics hash of the <code>Puppet::Transaction::Report</code> .
label	string	The name of the metric formatted as a title. Underscores are replaced with spaces and the first word is capitalized.
values	array	All the metric values within this category. Each value is in the form <code>[name, label, value]</code> , where <code>name</code> is the particular metric as a string, <code>label</code> is the metric name formatted as a title, and <code>value</code> is the metric quantity as an integer or a float.

The metrics that appear in a report are part of a fixed set and arranged in the following categories:

time

Includes a metric for every resource type for which there is at least one resource in the catalog, plus two additional metrics: `config_retrieval` and `total`. Each value in the `time` category is a float.

In an inspect report, there is an additional `inspect` metric.

resources

Includes the metrics `failed`, `out_of_sync`, `changed`, and `total`. Each value in the `resources` category is an integer.

events

Includes up to five metrics: `success`, `failure`, `audit`, `noop`, and `total`. `total` is always present; the others are present when their values are non-zero. Each value in the `events` category is an integer.

changes

Includes one metric, `total`. Its value is an integer.

Note: Failed reports contain no metrics.

Puppet::Resource::Status

A `Puppet::Resource::Status` object represents the status of a single resource.

Property	Type	Description
<code>resource_type</code>	string	The resource type, capitalized.
<code>title</code>	title	The resource title.
<code>resource</code>	string	The resource name, in the form <code>Type[title]</code> . This is always the same as the key that corresponds to this <code>Puppet::Resource::Status</code> object in the <code>resource_statuses</code> hash. Deprecated.
<code>provider_used</code>	string	The name of the provider used by the resource.
<code>file</code>	string	The path and filename of the manifest file that declared the resource.
<code>line</code>	integer	The line number in the manifest file that declared the resource.
<code>evaluation_time</code>	float	The amount of time, in seconds, taken to evaluate the resource. Not present in inspect reports.
<code>change_count</code>	integer	The number of properties that changed. Always 0 in inspect reports.
<code>out_of_sync_count</code>	integer	The number of properties that were out of sync. Always 0 in inspect reports.
<code>tags</code>	array	The strings with which the resource is tagged.
<code>time</code>	datetime	The time at which the resource was evaluated.
<code>events</code>	array	The <code>Puppet::Transaction::Event</code> objects for the resource.
<code>out_of_sync</code>	Boolean	True when <code>out_of_sync_count</code> > 0, otherwise false. Deprecated.

Property	Type	Description
changed	Boolean	True when <code>change_count > 0</code> , otherwise false. Deprecated.
skipped	Boolean	True when the resource was skipped, otherwise false.
failed	Boolean	True when Puppet experienced an error while evaluating this resource, otherwise false. Deprecated.
failed_to_restart	Boolean	True when Puppet experienced an error while trying to restart this resource, for example, when a Service resource has been notified from another resource.
containment_path	array	An array of strings; each element represents a container (type or class) that, together, make up the path of the resource in the catalog.

Puppet::Transaction::Event

A `Puppet::Transaction::Event` object represents a single event for a single resource.

Property	Type	Description
audited	Boolean	True when this property is being audited, otherwise false. True in inspect reports.
property	string	The property for which the event occurred. This value is missing if the provider errored out before it could be determined.
previous_value	string, array, or hash	The value of the property before the change (if any) was applied. This value is missing if the provider errored out before it could be determined.
desired_value	string, array, or hash	The value specified in the manifest. Absent in inspect reports. This value is missing if the provider errored out before it could be determined.
historical_value	string, array, or hash	The audited value from a previous run of Puppet, if known. Otherwise nil. Absent in inspect reports. This value is missing if the provider errored out before it could be determined.
message	string	The log message generated by this event.
name	symbol	The name of the event. Absent in inspect reports.

Property	Type	Description
status	string	<p>The event status:</p> <ul style="list-style-type: none"> • <code>success</code>: Property was out of sync and was successfully changed to be in sync. • <code>failure</code>: Property was out of sync and couldn't be changed to be in sync due to an error. • <code>noop</code>: Property was out of sync but wasn't changed because the run was in no-operation mode. • <code>audit</code>: Property was in sync and was being audited. Inspect reports are always in <code>audit</code> status.
redacted	Boolean	Whether this event has been redacted.
time	datetime	The time at which the property was evaluated.
corrective_change	Boolean	True if this event was caused by an unexpected change to the system between Puppet runs.

Differences from report format 9

- `failed_to_restart` was added to `Puppet::Resource::Status`

Puppet's internals

Learn the details of Puppet's internals, including how primary servers and agents communicate via host-verified HTTPS, and about the process of catalog compilation.

- [Agent-server HTTPS communications](#) on page 341

The Puppet agent and primary server communicate via mutually authenticated HTTPS using client certificates.

- [Catalog compilation](#) on page 342

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

Agent-server HTTPS communications

The Puppet agent and primary server communicate via mutually authenticated HTTPS using client certificates.

The HTTPS endpoints that Puppet uses are documented in the [HTTP API reference](#). Access to each endpoint is controlled by `auth.conf` settings. For more information, see [Puppet Server configuration files: auth.conf](#).

Persistent HTTP and HTTPS connections and Keep-Alive

When acting as an HTTPS client, Puppet reuses connections by sending `Connection: Keep-Alive` in HTTP requests. This reduces transport layer security (TLS) overhead, improving performance for runs with dozens of HTTPS requests.

You can configure the Keep-Alive duration using the `http_keepalive_timeout` setting, but it must be shorter than the maximum `keepalive` allowed by the primary server's web server.

Puppet caches HTTP connections and verified HTTPS connections. If you specify a custom HTTP connection class, Puppet does not cache the connection.

Puppet always requests that a connection is kept open, but the server can choose to close the connection by sending `Connection: close` in the HTTP response. If that occurs, Puppet does not cache the connection and starts a new connection for its next request.

For more information about the `http_keepalive_timeout` setting, see the [Configuration reference](#).

For an example of a server disabling persistent connections, see the [Apache documentation on KeepAlive](#).

The process of Agent-side checks and HTTPS requests during a single Puppet run.

1. Check for keys and certificates:

- a. The agent downloads the CA (Certification Authority) bundle.
- b. If certificate revocation is enabled, the agent loads or downloads the Certificate Revocation List (CRL) bundle using the previous CA bundle to verify the connection.
- c. The agent loads or generates a private key. If the agent needs a certificate, it generates a Certificate Signing Request (CSR), including any `dns_alt_names` and `csr_attributes`, and submits the request using `PUT /puppet-ca/v1/certificate_request/:certname`.
- d. The agent attempts to download the signed certificate using `GET /puppet-ca/v1/certificate/:certname`.
 - If there is a conflict that must be resolved on the Puppet server, such as cleaning the old CSR or certificate, the agent sleeps for `waitforcert` seconds, or exits with 1 if waiting is not allowed, such as when running `puppet agent -t`.

Tip: This can happen if the agent's SSL directory is deleted, as the Puppet server still has the valid, unrevoked certificate.

- If the downloaded certificate fails verification, such as it does not match its private key, then Puppet discards the certificate. The agent sleeps for `waitforcert` seconds, or exits with 1 if waiting is not allowed, such as when running `puppet agent -t`.

2. Request a node object and switch environments:

- Do a GET request to `/puppet/v3/node/<NAME>` .
 - If the request is successful, read the environment from the node object. If the node object has an environment, use that environment instead of the one in the agent's config file in all subsequent requests during this run.
 - If the request is unsuccessful, or if the node object had no environment set, use the environment from the agent's config file.

3. If `pluginsync` is enabled on the agent, fetch plugins from a file server mountpoint that scans the `lib` directory of every module:

- Do a GET request to `/puppet/v3/file_metadatas/plugins` with `recurse=true` and `links=manage`.
- Check whether any of the discovered plugins need to be downloaded. If so, do a GET request to `/puppet/v3/file_content/plugins/<FILE>` for each one.

4. Request catalog while submitting facts:

- Do a POST request to `/puppet/v3/catalog/<NAME>`, where the post data is all of the node's facts encoded as JSON. Receive a compiled catalog in return.

Note: Submitting facts isn't logically bound to requesting a catalog. For more information about facts, see [Language: Facts and built-in variables](#).

5. Make file source requests while applying the catalog:

File resources can specify file contents as either a `content` or `source` attribute. Content attributes go into the catalog, and the agent needs no additional data. Source attributes put only references into the catalog and might require additional HTTPS requests.

- If you are using the normal compiler, then for each file source, the agent makes a GET request to `/puppet/v3/file_metadata/<SOMETHING>` and compares the metadata to the state of the file on disk.
 - If it is in sync, it continues on to the next file source.
 - If it is out of sync, it does a GET request to `/puppet/v3/file_content/<SOMETHING>` for the content.
- If you are using the static compiler, all file metadata is embedded in the catalog. For each file source, the agent compares the embedded metadata to the state of the file on disk.
 - If it is in sync, it continues on to the next file source.
 - If it is out of sync, it does a GET request to `/puppet/v3/file_bucket_file/md5/<CHECKSUM>` for the content.

Note: Using a static compiler is more efficient with network traffic than using the normal (dynamic) compiler. Using the dynamic compiler is less efficient during catalog compilation. Large amounts of files, especially recursive directories, amplifies either issue.

6. If `report` is enabled on the agent, submit the report:

- Do a PUT request to `/puppet/v3/report/<NAME>`. The content of the PUT should be a Puppet report object in YAML format.

Catalog compilation

When configuring a node, the agent uses a document called a catalog, which it downloads from the primary server. 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 primary server and gives the agent a specific catalog.

This allows Puppet to:

- Separate privileges, because each node receives only its own resources.

- Reduce the agent's CPU and memory consumption.
- Simulate changes by running the agent in no-op mode, checking the agent's current state and reporting what would have changed without making any changes.
- Query PuppetDB for information about managed resources on any node.

Note: The `puppet apply` command compiles the catalog on its own node and then applies it, so it plays the role of both primary server and agent. To compile a catalog on the primary server for testing, run `puppet catalog compile` on the `puppetserver` with access to your environments, modules, manifests, and Hieradata.

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-server 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 primary server:

- 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 primary server. If the primary server 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 Hieradata 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 primary server.

1. Retrieve the node object.
 - After the primary server has received the agent-provided information for this request, it asks its configured node terminus for a node object.
 - By default, the primary server uses the `plain` node terminus, which returns a blank node object. In this case, only manifests and agent-provided information are used in compilation.
 - The next most common node terminus is the `exec` node terminus, which requests data from an ENC. This can return classes, variables, an environment, or a combination of the three, depending on how the ENC is designed.
 - You can also write a custom node terminus that retrieves classes, variables, and environments from an external system.
2. Set variables from the node object, from facts, and from the certificate.
 - All of these variables are available for use by any manifest or template during subsequent stages of compilation.
 - The node's facts are set as top-scope variables.
 - The node's facts are set in the protected `$facts` hash, and certain data from the node's certificate is set in the protected `$trusted` hash.
 - Any variables provided by the primary server 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 primary server 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 classes are defined in modules, although the main manifest can also contain class definitions.

 - If a matching node definition is found, the code in it is evaluated at node scope, overriding any top-scope variables. Resources in the code are added to the 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 primary server serves each environment with its own `modulepath`. When a class is loaded, the Puppet code in it is evaluated, and any resources in it are added to the catalog. If it was declared at node scope, it has access to node-scope variables; otherwise, it has access to only top-scope variables. Classes can also declare other classes; if they do, Puppet loads and evaluates those in the same way.
5. Evaluate classes from the node object
 - Puppet loads from modules and evaluate any classes that were specified by the node object. Resources from those classes are added to the catalog. If a matching node definition was found when the main manifest was evaluated, these classes are evaluated at node scope, which means that they can access any node-scope variables set by the main manifest. If no node definitions were present in the main manifest, they are evaluated at top scope.

Related information

[Classifying nodes](#) on page 329

You can classify nodes using an external node classifier (ENC), which is a script or application that tells Puppet which classes a node must have. It can replace or work in concert with the node definitions in the main site manifest (`site.pp`).

[Scope](#) on page 586

A scope is a specific area of code that is partially isolated from other areas of code.

[Node definitions](#) on page 441

A node definition, also known as a node statement, is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.

[The modulepath](#) on page 299

The primary server service and the `puppet apply` command load most of their content from modules found in one or more directories. The list of directories where Puppet looks for modules is called the *modulepath*. The modulepath is set by the current node's environment.

[Exported resources](#) on page 579

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.

Developing Puppet code

- [The Puppet language](#) on page 345

You'll use Puppet's declarative language to describe the desired state of your system.

- [Modules](#) on page 592

Modules manage a specific technology in your infrastructure and serve as the basic building blocks of Puppet desired state management.

- [Designing system configs: roles and profiles](#) on page 641

Your typical goal with Puppet is to build complete system configurations, which manage all of the software, services, and configuration that you care about on a given system. The roles and profiles method can help keep complexity under control and make your code more reusable, reconfigurable, and refactorable.

- [Puppet Forge](#) on page 668

Puppet Forge is a collection of modules and how-to guides developed by Puppet and its community.

- [Puppet Development Kit \(PDK\)](#) on page 668

You can write your own Puppet code and modules using Puppet Development Kit (PDK), which is a framework to successfully build, test and validate your modules.

- [Puppet VSCode extension](#) on page 668

Puppet has an extension for Visual Studio Code (VSCode) — Microsoft's cross-platform source-code editor.

The Puppet language

You'll use Puppet's declarative language to describe the desired state of your system.

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

Several parts of the Puppet language depend on evaluation order. For example, variables must be set before they are referenced. Throughout the language reference, we call out areas where the order of statements matters.

If you are new to Puppet, start with the Puppet language overview.

- [Puppet language overview](#) on page 348

The following overview covers some of the key components of the Puppet language, including catalogs, resources, classes and manifests.

- [Puppet language syntax examples](#) on page 350

A quick reference of syntax examples for the Puppet language.

- [The Puppet language style guide](#) on page 355

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

- [Files and paths on Windows](#) on page 379

Puppet and Windows handle directory separators and line endings in files somewhat differently, so you must be aware of the differences when you are writing manifests to manage Windows systems.

- [Code comments](#) on page 380

To add comments to your Puppet code, use shell-style or hash comments.

- [Variables](#) on page 380

Variables store values so that those values can be accessed in code later.

- [Resources](#) on page 383

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

- [Complete resource example](#)

- [Resource types](#) on page 391

Every resource (file, user, service, package, and so on) is associated with a resource type within the Puppet language. The resource type defines the kind of configuration it manages. This section provides information about the resource types that are built into Puppet.

- [Relationships and ordering](#) on page 405

Resources are included and applied in the order they are defined in their manifest, but only if the resource has no implicit relationship with another resource, as this can affect the declared order. To manage a group of resources in a specific order, explicitly declare such relationships with relationship metaparameters, chaining arrows, and the `require` function.

- [Classes](#) on page 411

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

- [Defined resource types](#) on page 418

Defined resource types, sometimes called defined types or defines, are blocks of Puppet code that can be evaluated multiple times with different parameters.

- [Bolt tasks](#) on page 421

Bolt tasks are single actions that you can run on target nodes in your infrastructure, allowing you to make as-needed changes to remote systems. You can run tasks with the Puppet Enterprise (PE) orchestrator or with Puppet's standalone task runner, Bolt.

- [Expressions and operators](#) on page 422

Expressions are statements that resolve to values. You can use expressions almost anywhere a value is required. Expressions can be compounded with other expressions, and the entire combined expression resolves to a single value.

- [Conditional statements and expressions](#) on page 431

Conditional statements let your Puppet code behave differently in different situations. They are most helpful when combined with facts or with data retrieved from an external source. Puppet supports *if* and *unless* statements, *case* statements, and *selectors*.

- [Function calls](#) on page 438

Functions are plug-ins, written in Ruby, that you can call during catalog compilation. A call to any function is an expression that resolves to a value. Most functions accept one or more values as arguments, and return a resulting value.

- [Built-in functions](#)

- [Node definitions](#) on page 441

A node definition, also known as a node statement, is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.

- [Facts and built-in variables](#) on page 443

Before requesting a catalog for a managed node, or compiling one with `puppet apply`, Puppet collects system information, called *facts*, by using the *Facter* tool. The facts are assigned as values to variables that you can use anywhere in your manifests. Puppet also sets some additional special variables, called *built-in variables*, which behave a lot like facts.

- [Reserved words and acceptable names](#) on page 449

You can use only certain characters for naming variables, modules, classes, defined types, and other custom constructs. Additionally, some words in the Puppet language are reserved and cannot be used as bare word strings or names.

- [Custom resources](#) on page 454

A *resource* is the basic unit that is managed by Puppet. Each resource has a set of attributes describing its state. Some attributes can be changed throughout the lifetime of the resource, whereas others are only reported back but cannot be changed, and some can only be set one time during initial creation.

- [Custom functions](#) on page 483

Use the Puppet language, or the Ruby API to create custom functions.

- [Values, data types, and aliases](#) on page 504

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

- [Templates](#) on page 553

Templates are written in a specialized templating language that generates text from data. Use templates to manage the content of your Puppet configuration files via the `content` attribute of the `file` resource type.

- [Advanced constructs](#) on page 569

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

- [Details of complex behaviors](#) on page 583

Within Puppet language there are complex behavior patterns regarding classes, defined types, and specific areas of code called scopes.

Related information

[Puppet language overview](#) on page 348

The following overview covers some of the key components of the Puppet language, including catalogs, resources, classes and manifests.

[Puppet language syntax examples](#) on page 350

A quick reference of syntax examples for the Puppet language.

[The Puppet language style guide](#) on page 355

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.

Puppet language overview

The following overview covers some of the key components of the Puppet language, including catalogs, resources, classes and manifests.

The following video gives you an overview of the Puppet language:

Related information

[The Puppet language](#) on page 345

You'll use Puppet's declarative language to describe the desired state of your system.

[Puppet language syntax examples](#) on page 350

A quick reference of syntax examples for the Puppet language.

[The Puppet language style guide](#) on page 355

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

Catalogs

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

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

To create a catalog for a given agent, the primary server compiles:

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

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

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

Agents cache their most recent catalog. If they request a catalog and the primary server fails to compile one, they fall back to their cached catalog. For detailed information on the catalog compilation process, see the [catalog compilation system](#) page.

Related information

[Catalog compilation](#) on page 342

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

Resources and classes

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

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

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

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

The following video gives you an overview of resources:

The following video gives you an overview of classes:

Related information

[Resources](#) on page 383

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

[Classes](#) on page 411

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

Manifests

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

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

Manifests:

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

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

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

Manifest example

This short manifest manages NTP. It includes:

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

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

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

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

file { 'ntp.conf':
  path      => '/etc/ntp.conf',
  ensure    => file,
  require   => Package['ntp'],
  source    => "puppet:///modules/ntp/ntp.conf",
  # This source file would be located on the primary Puppet server at
  # /etc/puppetlabs/code/modules/ntp/files/ntp.conf
}
```

Puppet language syntax examples

A quick reference of syntax examples for the Puppet language.

Resource examples

Resource declaration

This example resource declaration includes:

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

```
file { 'ntp.conf':
  path      => '/etc/ntp.conf',
```

```

ensure => file,
content => template('ntp/ntp.conf'),
owner  => 'root',
mode   => '0644',
}

```

For details about resources and resource declaration syntax, see [Resources](#) on page 383.

Resource relationship metaparameters

Two resource declarations establishing relationships with the `before` and `subscribe` *metaparameters*, which accept resource references.

The first declaration ensures that the `ntp` package is installed before the `ntp.conf` file is created. The second declaration ensures that the `ntpd` service is notified of any changes to the `ntp.conf` file.

```

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

```

For details about relationships usage and syntax, see [Relationships and ordering](#) on page 405. For details about resource references, see [Resource and class references](#) on page 535.

Resource relationship chaining arrows

Chaining arrows forming relationships between three resources, using resource references. In this example, the `ntp` package must be installed before the `ntp.conf` file is created; after the file is created, the `ntpd` service is notified.

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

For details about relationships usage and syntax, see [Relationships and ordering](#) on page 405. For details about resource references, see [Resource and class references](#) on page 535.

Exported resource declaration

An *exported resource* declaration.

```

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

```

For information about declaring and collecting exported resources, see [Exported resources](#).

Resource collector

A *resource collector*, sometimes called the "spaceship operator." Resource collectors select a group of resources by searching the attributes of each resource in the catalog.

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

For details about resource collector usage and syntax, see [Resource collectors](#).

Exported resource collector

An exported resource collector, which works with *exported resources*, which are available for use by other nodes.

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

For details about resource collector usage and syntax, see [Resource collectors](#). For information about declaring and collecting exported resources, see [Exported resources](#).

Resource default for the `exec` type

A resource default statement set default attribute values for a given resource type. This example specifies defaults for the `exec` resource type attributes `path`, `environment`, `logoutput`, and `timeout`.

```
Exec {
  path      => '/usr/bin:/bin:/usr/sbin:/sbin',
  environment => 'RUBYLIB=/opt/puppetlabs/puppet/lib/ruby/site_ruby/2.1.0/',
  logoutput  => true,
  timeout    => 180,
}
```

For details about default statement usage and syntax, see [Resource defaults](#).

Virtual resource

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

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

For details about virtual resource usage and syntax, see [Virtual resources](#).

Defined resource type examples

Defined resource type definition

Defining a type creates a new defined resource type. The name of this *defined type* has two namespace segments, comprising the name of the module containing the defined type, `apache`, and the name of the defined type itself, `vhost`.

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


For details about defined type usage and syntax, see [Defined resource types](#) on page 418.

Defined type resource declaration

Declarations of an instance, or resource, of a defined type are similar to other resource declarations. This example declares a instance of the `apache::vhost` defined type, with a title of "homepages" and the `port` and `docroot` attributes specified.

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

For details about defined type usage and syntax, see [Defined resource types](#) on page 418.

Defined type resource reference

A resource reference to an instance of the `apache::vhost` defined resource. Every namespace segment in a resource reference must be capitalized.

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

For details about defined type usage and syntax, see [Defined resource types](#) on page 418. For details about resource references, see [Resource and class references](#) on page 535.

Class examples

Class definition

A class definition, which makes a class available for later use.

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

For details about class usage and syntax, see [Classes](#) on page 411.

Class declaration

Declaring the `ntp` class in three different ways:

- the `include` function
- the `require` function
- the resource-like syntax

Declaring a class causes the resources in it to be managed.

The `include` function is the standard way to declare classes:

```
include ntp
```

The `require` function declares the class and makes it a dependency of the code container where it is declared:

```
require ntp
```

The resource-like syntax declares the class and applies resource-like behavior. Resource-like class declarations require that you declare a given class only one time.

```
class { 'ntp': }
```

For details about class usage and syntax, see [Classes](#) on page 411.

Variable examples

Variable assigned an array value

A variable being assigned a set of values as an array.

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

For details about assigning values to variables, see [Variables](#) on page 380.

Variable assigned a hash value

A variable being assigned a set of values as a hash.

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

For details about assigning values to variables, see [Variables](#) on page 380.

Interpolated variable

A built-in variable provided by the primary server being interpolated into a double-quoted string.

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

For details about built-in variables usage and syntax, see [Facts and built-in variables](#) on page 443. For information about strings and interpolation, see [Strings](#) on page 542.

Conditional statement examples

if statement, using expressions and facts

An if statement, whose conditions are expressions that use facts provided by the agent.

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

For details about if statements, see [Conditional statements and expressions](#) on page 431.

if statement, with in expression

An if statement using an in expression.

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

```
}
```

For details about `if` statements, see [Conditional statements and expressions](#) on page 431.

Case statement

A case statement.

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

For details about case statements, see [Conditional statements and expressions](#) on page 431.

Selector statement

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

```
$rootgroup = $osfamily ? {
  'RedHat'           => 'wheel',
  /(Debian|Ubuntu)/ => 'wheel',
  default            => 'root',
}
```

For details about selector statements, see [Conditional statements and expressions](#) on page 431.

Node examples

Node definition

A node definition or node statement is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.

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

Node names in node definitions can also be given as regular expressions.

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

For details about node definition usage and syntax, see [Node definitions](#) on page 441.

The Puppet language style guide

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

This style guide applies to Puppet 4 and later. Puppet 3 is no longer supported, but we include some Puppet 3 guidelines in case you're maintaining older code.

Tip: Use [puppet-lint](#) and [metadata-json-lint](#) to check your module for compliance with the style guide.

No style guide can cover every circumstance you might run into when developing Puppet code. When you need to make a judgement call, keep in mind a few general principles.

Readability matters

If you have to choose between two equal alternatives, pick the more readable one. This is subjective, but if you can read your own code three months from now, it's a great start. In particular, code that generates readable diffs is highly preferred.

Scoping and simplicity are key

When in doubt, err on the side of simplicity. A module should contain related resources that enable it to accomplish a task. If you describe the function of your module and you find yourself using the word "and," consider splitting the module. Have one goal, with all your classes and parameters focused on achieving it.

Your module is a piece of software

At least, we recommend that you treat it that way. When it comes to making decisions, choose the option that is easier to maintain in the long term.

These guidelines apply to Puppet code, for example, code in Puppet modules or classes. To reduce repetitive phrasing, we don't include the word 'Puppet' in every description, but you can assume it.

For information about the specific meaning of terms like 'must,' 'must not,' 'required,' 'should,' 'should not,' 'recommend,' 'may,' and 'optional,' see [RFC 2119](#).

Module design practices

Consistent module design practices makes module contributions easier.

Spacing, indentation, and whitespace

Module manifests should follow best practices for spacing, indentation, and whitespace.

Manifests:

- Must use two-space soft tabs.
- Must not use literal tab characters.
- Must not contain trailing whitespace.
- Must include trailing commas after all resource attributes and parameter definitions.
- Must end the last line with a new line.
- Must use one space between the resource type and opening brace, one space between the opening brace and the title, and no spaces between the title and colon.

Good:

```
file { '/tmp/sample':
```

Bad: Space between title and colon:

```
file { '/tmp/sample' :
```

Bad: No spaces:

```
file{'/tmp/sample':
```

Bad: Too many spaces:

```
file      { '/tmp/sample':
```

- Should not exceed a 140-character line width, except where such a limit would be impractical.
- Should leave one empty line between resources, except when using dependency chains.
- May align hash rockets (=>) within blocks of attributes, one space after the longest resource key, arranging hashes for maximum readability first.

Arrays and hashes

To increase readability of arrays and hashes, it is almost always beneficial to break up the elements on separate lines.

Use a single line only if that results in overall better readability of the construct where it appears, such as when it is very short. When breaking arrays and hashes, they should have:

- Each element on its 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 an indentation of elements past two spaces:

```
service { 'sshd':
  require => [
    Package['openssh-server'],
    File['/etc/ssh/sshd_config'],
  ],
}
```

Quoting

As long as you are consistent, strings may be enclosed in single or double quotes, depending on your preference.

Regardless of your preferred quoting style, all variables **MUST** be enclosed in braces when interpolated in a string.

For example:

Good:

```
"/etc/${file}.conf"
```

```
"${facts['operatingsystem']} is not supported by ${module_name}"
```

Bad:

```
"/etc/$file.conf"
```

Option 1: Prefer single quotes

Modules that adopt this string quoting style **MUST** enclose all strings in single quotes, except as listed below.

For example:

Good:

```
owner => 'root'
```

Bad:

```
owner => "root"
```

A string **MUST** be enclosed in double quotes if it:

- Contains variable interpolations.

- Good:

```
"/etc/${file}.conf"
```

- Bad:

```
'/etc/${file}.conf'
```

- Contains escaped characters not supported by single-quoted strings.

- Good:

```
content => "nameserver 8.8.8.8\n"
```

- Bad:

```
content => 'nameserver 8.8.8.8\n'
```

A string **SHOULD** be enclosed in double quotes if it:

- Contains single quotes.

- Good:

```
warning("Class['apache'] parameter purge_vdir is deprecated in favor of
purge_configs")
```

- Bad:

```
warning('Class[\'apache\'] parameter purge_vdir is deprecated in favor
of purge_configs')
```

Option 2: Prefer double quotes

Modules that adopt this string quoting style **MUST** enclose all strings in double quotes, except as listed below.

For example:

Good:

```
owner => "root"
```

Bad:

```
owner => 'root'
```

A string **SHOULD** be enclosed in single quotes if it does not contain variable interpolations **AND** it:

- Contains double quotes.

- Good:

```
warning('Class["apache"] parameter purge_vdir is deprecated in favor of
purge_configs')
```

- Bad:

```
warning("Class[\'apache\'] parameter purge_vdir is deprecated in favor
of purge_configs")
```

- Contains literal backslash characters that are not intended to be part of an escape sequence.

- Good:

```
path => 'c:\windows\system32'
```

- Bad:

```
path => "c:\\windows\\system32"
```

If a string is a value from an enumerable set of options, such as `present` and `absent`, it **SHOULD NOT** be enclosed in quotes at all.

For example:

Good:

```
ensure => present
```

Bad:

```
ensure => "present"
```

Escape characters

Use backslash (\) as an escape character.

For both single- and double-quoted strings, escape the backslash to remove this special meaning: \\ This means that for every backslash you want to include in the resulting string, use two backslashes. As an example, to include two literal backslashes in the string, you would use four backslashes in total.

Do not rely on unrecognized escaped characters as a method for including the backslash and the character following it.

Unicode character escapes using fewer than 4 hex digits, as in \u040, results in a backslash followed by the string u040. (This also causes a warning for the unrecognized escape.) To use a number of hex digits not equal to 4, use the longer u{digits} format.

Comments

Comments must be hash comments (# This is a comment). Comments should explain the why, not the how, of your code.

Do not use /* */ comments in Puppet code.

Good:

```
# Configures NTP
file { '/etc/ntp.conf': ... }
```

Bad:

```
/* Creates file /etc/ntp.conf */
file { '/etc/ntp.conf': ... }
```

Note: Include documentation comments for Puppet Strings for each of your classes, defined types, functions, and resource types and providers. If used, documentation comments precede the name of the element. For documentation recommendations, see the Modules section of this guide.

Functions

Avoid the `inline_template()` and `inline_epp()` functions for templates of more than one line, because these functions don't permit template validation. Instead, use the `template()` and `epp()` functions to read a template from the module. This method allows for syntax validation.

You should avoid using calls to Hiera functions in modules meant for public consumption, because not all users have implemented Hiera. Instead, we recommend using parameters that can be overridden with Hiera.

Improving readability when chaining functions

In most cases, especially if blocks are short, we recommend keeping functions on the same line. If you have a particularly long chain of operations or block that you find difficult to read, you can break it up on multiples lines to improve readability. As long as your formatting is consistent throughout the chain, it is up to your own judgment.

For example, this:

```
$foodgroups.fruit.vegetables
```

Is better than this:

```
$foodgroups
  .fruit
  .vegetables
```


But, this:

```
$foods = {
  "avocado"    => "fruit",
  "eggplant"   => "vegetable",
  "strawberry" => "fruit",
  "raspberry"  => "fruit",
}

$berries = $foods.filter |$name, $kind| {
  # Choose only fruits
  $kind == "fruit"
}.map |$name, $kind| {
  # Return array of capitalized fruits
  String($name, "%c")
}.filter |$fruit| {
  # Only keep fruits named "berry"
  $fruit =~ /berry$/
}
```

Is better than this:

```
$foods = {
  "avocado"    => "fruit",
  "eggplant"   => "vegetable",
  "strawberry" => "fruit",
  "raspberry"  => "fruit",
}

$berries = $foods.filter |$name, $kind| { $kind == "fruit" }.map |$name,
  $kind| { String($name, "%c") }.filter |$fruit| { $fruit =~ /berry$/ }
```

Related information

[Modules](#) on page 376

Develop your module using consistent code and module structures to make it easier to update and maintain.

Resources

Resources are the fundamental unit for modeling system configurations. Resource declarations have a lot of possible features, so your code's readability is crucial.

Resource names

All resource names or titles must be quoted. If you are using an array of titles you must quote each title in the array, but cannot quote the array itself.

Good:

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

Bad:

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

These quoting requirements do not apply to expressions that evaluate to strings.

Arrow alignment

To align hash rockets (=>) in a resource's attribute/value list or in a nested block, place the hash rocket one space ahead of the longest attribute name. Indent the nested block by two spaces, and place each attribute on a separate line. Declare very short or single purpose resource declarations on a single line.

Good:

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

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

myresource { 'test':
  ensure => present,
  myhash => {
    'myhash_key1' => 'value1',
    'key2'        => 'value2',
  },
}

notify { 'warning': message => 'This is an example warning' }
```

Bad:

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

file { "/path/to/my-filename.txt":
  ensure => file, mode => $mode, owner => $owner, group => $group,
  source => 'puppet:///modules/my-module/productions/my-filename.txt'
}
```

Attribute ordering

If a resource declaration includes an `ensure` attribute, it should be the first attribute specified so that a user can quickly see if the resource is being created or deleted.

Good:

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

When using the special attribute `*` (asterisk or splat character) in addition to other attributes, splat should be ordered last so that it is easy to see. You may not include multiple splats in the same body.

Good:

```
$file_ownership = {
  'owner' => 'root',
  'group' => 'wheel',
  'mode'  => '0644',
}

file { '/etc/passwd':
  ensure => file,
  *      => $file_ownership,
```

```
}
```

Resource arrangement

Within a manifest, resources should be grouped by logical relationship to each other, rather than by resource type.

Good:

```
file { ['/tmp/dir':
  ensure => directory,
}

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

file { ['/tmp/dir2':
  ensure => directory,
}

file { ['/tmp/dir2/b':
  content => 'b',
}
```

Bad:

```
file { ['/tmp/dir':
  ensure => directory,
}

file { ['/tmp/dir2':
  ensure => directory,
}

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

file { ['/tmp/dir2/b':
  content => 'b',
}
```

Use semicolon-separated multiple resource bodies only in conjunction with a local default body.

Good:

```
$defaults = { < hash of defaults > }

file {
  default:
    * => $defaults,;

  '/tmp/testfile':
    content => 'content of the test file',
}
```

Good: Repeated pattern with defaults:

```
$defaults = { < hash of defaults > }

file {
  default:
```

```

    * => $defaults,;

    '/tmp/motd':
      content => 'message of the day',;

    '/tmp/motd_tomorrow':
      content => 'tomorrows message of the day',;
  }

```

Bad: Unrelated resources grouped:

```

file {
  '/tmp/testfile':
    owner   => 'admin',
    mode    => '0644',
    contents => 'this is the content',;

  '/opt/myapp':
    owner   => 'myapp-admin',
    mode    => '0644',
    source  => 'puppet://<someurl>',;

  # etc
}

```

You cannot set any attribute more than one time for a given resource; if you try, Puppet raises a compilation error. This means:

- If you use a hash to set attributes for a resource, you cannot set a different, explicit value for any of those attributes. For example, if mode is present in the hash, you can't also set `mode => "0644"` in that resource body.
- You can't use the `*` attribute multiple times in one resource body, because `*` itself acts like an attribute.
- To use some attributes from a hash and override others, either use a hash to set per-expression defaults, or use the `+` (merging) operator to combine attributes from two hashes (with the right-hand hash overriding the left-hand one).

Symbolic links

Declare symbolic links with an ensure value of `ensure => link`. To inform the user that you are creating a link, specify a value for the `target` attribute.

Good:

```

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

```

Bad:

```

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

```

File modes

- POSIX numeric notation must be represented as 4 digits.
- POSIX symbolic notation must be a string.
- You should not use file mode with Windows; instead use the [acl module](#).
- You should use numeric notation whenever possible.

- The file mode attribute should always be a quoted string or (unquoted) variable, never an integer.

Good:

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

Bad:

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

Multiple resources

Multiple resources declared in a single block should be used only when there is also a default set of options for the resource type.

Good:

```
file {
  default:
    ensure => 'file',
    mode   => '0666',;

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

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

Good: Give the defaults a name if used several times:

```
$our_default_file_attributes = {
  'ensure' => 'file',
  'mode'   => '0666',
}

file {
  default:
    * => $our_default_file_attributes,;

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

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

Good: Spell out 'magic' iteration:

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

Good: Spell out 'magic' iteration:

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

Bad:

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

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

file { ['/owner', '/staff']:
  ensure => 'file',
}

file { $array_of_paths:
  ensure => 'file',
}
```

Legacy style defaults

Avoid legacy style defaults. If you do use them, they should occur only at top scope in your site manifest. This is because resource defaults propagate through dynamic scope, which can have unpredictable effects far away from where the default was declared.

Acceptable: site.pp:

```
Package {
  provider => 'zypper',
}
```

Bad: /etc/puppetlabs/puppet/modules/apache/manifests/init.pp:

```
File {
  owner => 'nobody',
  group => 'nogroup',
  mode  => '0600',
}

concat { $config_file_path:
  notify => Class['Apache::Service'],
  require => Package['httpd'],
}
```

Attribute alignment

Resource attributes must be uniformly indented in two spaces from the title.

Good:

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

Bad: Too many levels of indentation:

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

Bad: No indentation:

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

Bad: Improper and non-uniform indentation:

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

Bad: Indented the wrong direction:

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

For multiple bodies, each title should be on its own line, and be indented. You may align all arrows across the bodies, but arrow alignment is not required if alignment per body is more readable.

```
file {
  default:
    * => $local_defaults,;

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

Defined resource types

Because defined resource types can have multiple instances, resource names must have a unique variable to avoid duplicate declarations.

Good: Template uses `$listen_addr_port`:

```
define apache::listen {
  $listen_addr_port = $name

  concat::fragment { "Listen ${listen_addr_port}":
    ensure => present,
    target => $::apache::ports_file,
```

```

    content => template('apache/listen.erb'),
  }
}

```

Bad: Template uses \$name:

```

define apache::listen {
  concat::fragment { 'Listen port':
    ensure => present,
    target => $::apache::ports_file,
    content => template('apache/listen.erb'),
  }
}

```

Classes and defined types

Classes and defined types should follow scope and organization guidelines.

Separate files

Put all classes and resource type definitions (defined types) as separate files in the `manifests` directory of the module. Each file in the manifest directory should contain nothing other than the class or resource type definition.

Good: `etc/puppetlabs/puppet/modules/apache/manifests/init.pp`:

```

class apache { }

```

Good: `etc/puppetlabs/puppet/modules/apache/manifests/ssl.pp`:

```

class apache::ssl { }

```

Good: `etc/puppetlabs/puppet/modules/apache/manifests/virtual_host.pp`:

```

define apache::virtual_host () { }

```

Separating classes and defined types into separate files is functionally identical to declaring them in `init.pp`, but has the benefit of highlighting the structure of the module and making the function and structure more legible.

When a resource or include statement is placed outside of a class, node definition, or defined type, it is included in all catalogs. This can have undesired effects and is not always easy to detect.

Good: `manifests/init.pp`:

```

# class ntp
class ntp {
  ntp::install
}
# end of file

```

Bad: `manifests/init.pp`:

```

class ntp {
  #...
}
ntp::install

```

Internal organization of classes and defined types

Structure classes and defined types to accomplish one task.

Documentation comments for Puppet Strings should be included for each class or defined type. If used, documentation comments must precede the name of the element. For complete documentation recommendations, see the Modules section.

Put the lines of code in the following order:

1. First line: Name of class or type.
2. Following lines, if applicable: Define parameters. Parameters should be [typed](#).
3. Next lines: Includes and validation come after parameters are defined. Includes may come before or after validation, but should be grouped separately, with all includes and requires in one group and all validations in another. Validations should validate any parameters and fail catalog compilation if any parameters are invalid. See [puppetlabs-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'
```

Public and private

Split your module into public and private classes and defined types where possible. Public classes or defined types should contain the parts of the module meant to be configured or customized by the user, while private classes should contain things you do not expect the user to change via parameters. Separating into public and private classes or defined types helps build reusable and readable code.

Help indicate to the user which classes are which by making sure all public classes have complete comments and denoting public and private classes in your documentation. Use the documentation tags “@api private” and “@api public” to make this clear. For complete documentation recommendations, see the Modules section.

Chaining arrow syntax

Most of the time, use [relationship metaparameters](#) rather than [chaining arrows](#). When you have many [interdependent or order-specific items](#), chaining syntax may be used. A chain operator should appear on the same line as its right-hand operand. Chaining arrows must be used left to right.

Good: Points left to right:

```
Package['httpd'] -> Service['httpd']
```

Good: On the line of the right-hand operand:

```
Package['httpd']
-> Service['httpd']
```

Bad: Arrows are not all pointing to the right:

```
Service['httpd'] <- Package['httpd']
```

Bad: Must be on the right-hand operand's line:

```
Package['httpd'] ->
Service['httpd']
```

Nested classes or defined types

Don't define classes and defined resource types within other classes or defined types. Declare them as close to node scope as possible. If you have a class or defined type which requires another class or defined type, put graceful failures in place if those required classes or defined types are not declared elsewhere.

Bad:

```
class apache {
  class ssl { ... }
}
```

Bad:

```
class apache {
  define config() { ... }
}
```

Display order of parameters

In parameterized class and defined resource type definitions, you can list required parameters before optional parameters (that is, parameters with defaults). Required parameters are parameters that are not set to anything, including undef. For example, parameters such as passwords or IP addresses might not have reasonable default values.

You can also group related parameters, order them alphabetically, or in the order you encounter them in the code. How you order parameters is personal preference.

Note that treating a parameter like a namevar and defaulting it to `$title` or `$name` does not make it a required parameter. It should still be listed following the order recommended here.

Good:

```
class dhcp (
  $dnsdomain,
  $nameservers,
  $default_lease_time = 3600,
  $max_lease_time     = 86400,
) {}
```

Bad:

```
class ntp (
  $options = "iburst",
  $servers,
  $multicast = false,
) {}
```

Parameter defaults

Adding default values to the parameters in classes and defined types makes your module easier to use. Use Hiera data in your module to set parameter defaults. See [Defining classes](#) on page 411 for details about setting parameter defaults with Hiera data. In simple cases, you can also specify the default values directly in the class or defined type.

Be sure to declare the data type of parameters, as this provides automatic type assertions.

Good: Parameter defaults set in the class with references to Hiera data:

```
class my_module (
  String $source,
  String $config,
) {
```

```
# body of class
}
```

A `hiera.yaml` in the root of the module sets the hierarchy for assigning defaults:

```
---
version: 5
default_hierarchy:
- name: 'defaults'
  path: 'defaults.yaml'
  data_hash: yaml_data
```

And the file `data/defaults.yaml` specifies the actual default values:

```
my_module::source: 'default source value'
my_module::config: 'default config value'
```

This example places the values in the defaults hierarchy, which means that the defaults are not merged into overriding values. To merge the defaults into those values, change the `default_hierarchy` to `hierarchy`.

If you are maintaining old code created prior to Puppet 4.9, you might encounter the use of a `params.pp` pattern. This pattern makes maintenance and troubleshooting difficult — refactor such code to use the `Hiera data-in-modules` pattern instead. See [Adding Hiera data to a module](#) on page 286 for a detailed example showing how to replace `params.pp` with `data`.

Bad: `params.pp`

```
class my_module (
  String $source = $my_module::params::source,
  String $config = $my_module::params::config,
) inherits my_module::params {
  # body of class
}
```

Exported resources

Exported resources should be opt-in rather than opt-out. Your module should not be written to use exported resources to function by default unless it is expressly required.

When using exported resources, name the property `collect_exported`.

Exported resources should be exported and collected selectively using a [search expression](#), ideally allowing user-defined tags as parameters so tags can be used to selectively collect by environment or custom fact.

Good:

```
define haproxy::frontend (
  $ports          = undef,
  $ipaddress       = [${::ipaddress}],
  $bind            = undef,
  $mode            = undef,
  $collect_exported = false,
  $options         = {
    'option' => [
      'tcplog',
    ],
  },
) {
  # body of define
}
```

Parameter indentation and alignment

Parameters to classes or defined types must be uniformly indented in two spaces from the title. The equals sign should be aligned.

Good:

```
class profile::myclass (
  $var1    = 'default',
  $var2    = 'something else',
  $another = 'another default value',
) {
}
```

Good:

```
class ntp (
  Boolean                $broadcastclient = false,
  Optional[Stdlib::Absolutepath] $config_dir    = undef,
  Enum['running', 'stopped']    $service_ensure = 'running',
  String                 $package_ensure  = 'present',
  # ...
) {
# ...
}
```

Bad: Too many level of indentation:

```
class profile::myclass (
  $var1    = 'default',
  $var2    = 'something else',
  $another = 'another default value',
) {
}
```

Bad: No indentation:

```
class profile::myclass (
$var1    = 'default',
$var2    = 'something else',
$another = 'another default value',
) {
}
```

Bad: Misaligned equals sign:

```
class profile::myclass (
  $var1 = 'default',
  $var2 = 'something else',
  $another = 'another default value',
) {
}
```

Class inheritance

In addition to scope and organization, there are some additional guidelines for handling classes in your module.

Don't use class inheritance; use data binding instead of `params.pp` pattern. Inheritance is used only for `params.pp`, which is not recommended in Puppet 4.

If you use inheritance for maintaining older modules, do not use it across module namespaces. To satisfy cross-module dependencies in a more portable way, include statements or relationship declarations. Only use class inheritance for `myclass::params` parameter defaults. Accomplish other use cases by adding parameters or conditional logic.

Good:

```
class ssh { ... }

class ssh::client inherits ssh { ... }

class ssh::server inherits ssh { ... }
```

Bad:

```
class ssh inherits server { ... }

class ssh::client inherits workstation { ... }

class wordpress inherits apache { ... }
```

Public modules

When declaring classes in publicly available modules, use `include`, `contain`, or `require` rather than class resource declaration. This avoids duplicate class declarations and vendor lock-in.

Type signatures

We recommend always using type signatures for class and defined type parameters. Keep the parameters and `=` signs aligned.

When dealing with very long type signatures, you can define type aliases and use short definitions. Good naming of aliases can also serve as documentation, making your code easier to read and understand. Or, if necessary, you can turn the 140 line character limit off. For more information on type signatures, see [the Type data type](#).

Related information

[Modules](#) on page 376

Develop your module using consistent code and module structures to make it easier to update and maintain.

Variables

Reference variables in a clear, unambiguous way that is consistent with the Puppet style.

Referencing facts

When referencing facts, prefer the `$facts` hash to plain top-scope variables (such as `$::operatingsystem`).

Although plain top-scope variables are easier to write, the `$facts` hash is clearer, easier to read, and distinguishes facts from other top-scope variables.

Namespacing variables

When referencing top-scope variables other than facts, explicitly specify absolute namespaces for clarity and improved readability. This includes top-scope variables set by the node classifier and in the main manifest.

This is not necessary for:

- the `$facts` hash.
- the `$trusted` hash.

- the `$server_facts` hash.

These special variable names are protected; because you cannot create local variables with these names, they always refer to top-scope variables.

Good:

```
$facts['operatingsystem']
```

Bad:

```
$::operatingsystem
```

Very bad:

```
$operatingsystem
```

Variable format

When defining variables you must only use numbers, lowercase letters, and underscores. Do not use upper-case letters within a word, such as “CamelCase”, as it introduces inconsistency in style. You must not use dashes, as they are not syntactically valid.

Good:

```
$server_facts
$total_number_of_entries
$error_count123
```

Bad:

```
$serverFacts
$totalNumberOfEntries
$error-count123
```

Conditionals

Conditional statements should follow Puppet code guidelines.

Simple resource declarations

Avoid mixing conditionals with resource declarations. When you use conditionals for data assignment, separate conditional code from the resource declarations.

Good:

```
$file_mode = $facts['operatingsystem'] ? {
  'debian' => '0007',
  'redhat' => '0776',
  default => '0700',
}

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

Bad:

```
file { ['/tmp/readme.txt':
```

```

ensure => file,
content => "Hello World\n",
mode   => $facts['operatingsystem'] ? {
  'debian' => '0777',
  'redhat' => '0776',
  default  => '0700',
}
}

```

Defaults for case statements and selectors

Case statements must have default cases. If you want the default case to be "do nothing," you must include it as an explicit default: `{}` for clarity's sake.

Case and selector values must be enclosed in quotation marks.

Selectors should omit default selections only if you explicitly want catalog compilation to fail when no value matches.

Good:

```

case $facts['operatingsystem'] {
  'centos': {
    $version = '1.2.3'
  }
  'ubuntu': {
    $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.

Conditional statement alignment

When using if/else statements, align in the following way:

```

if $something {
  $var = 'hour'
} elsif $something_else {
  $var = 'minute'
} else {
  $var = 'second'
}

```

For more information on if/else statements, see [Conditional statements and expressions](#).

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.

You can find thorough, detailed information on writing a great README in [Documenting modules](#) on page 618, but in general your README should:

- Summarize what your module does.
- Note any setup requirements or limitations, such as "This module requires the `puppetlabs-apache` module and only works on Ubuntu."
- Note any part of a user's system the module might impact (for example, "This module overwrites everything in `animportantfile.conf`").
- Describe how to customize and configure the module.
- Include usage examples and code samples for the common use cases for your module.

Documenting Puppet code

Use [Puppet Strings](#) code comments to document your Puppet classes, defined types, functions, and resource types and providers. Strings processes the README and comments from your code into HTML or JSON format documentation. This allows you and your users to generate detailed documentation for your module.

Include comments for each element (classes, functions, defined types, parameters, and so on) in your module. If used, comments must precede the code for that element. Comments should contain the following information, arranged in this order:

- A description giving an overview of what the element does.
- Any additional information about valid values that is not clear from the data type. For example, if the data type is `[String]`, but the value must specifically be a path.
- The default value, if any, for that element,

Multiline descriptions must be uniformly indented by at least one space:

```
# @param config_epp Specifies a file to act as a EPP template for the config
# file.
# Valid options: a path (absolute, or relative to the module path). Example
# value:
# 'ntp/ntp.conf.epp'. A validation error is thrown if you supply both this
# param **and**
# the `config_template` param.
```

If you use Strings to document your module, include information about Strings in the Reference section of your README so that your users know how to generate the documentation. See [Puppet Strings](#) documentation for details on usage, installation, and correctly writing documentation comments.

If you do not include Strings code comments, you should include a Reference section in your README with a complete list of all classes, types, providers, defined types, and parameters that the user can configure. Include a brief description, the valid options, and the default values (if any).

For example, this is a parameter for the `ntp` module's `ntp` class: `package_ensure`:

```
Data type: String.

Whether to install the NTP package, and what version to install. Values:
'present', 'latest', or a specific version number.

Default value: 'present'.
```

For more details and examples, see the [module documentation guide](#).

CHANGELOG

Your module should include a change log file called `CHANGELOG.md` or `.markdown`. Your change log should:

- Have entries for each release.
- List bugfixes and features included in the release.
- Specifically call out backwards-incompatible changes.

Examples

In the `/examples` directory, include example manifests that demonstrate major use cases for your module.

```
modulepath/apache/examples/{usecase}.pp
```

The example manifest should provide a clear example of how to declare the class or defined resource type. It should also declare any classes required by the corresponding class to ensure `puppet apply` works in a limited, standalone manner.

Testing

Use one or more of the following community tools for testing your code and style:

- [puppet-lint](#) tests your code for adherence to the style guidelines.
- [metadata-json-lint](#) tests your `metadata.json` for adherence to the style guidelines.
- For testing your module, we recommend [rspec](#). [rspec-puppet](#) can help you write `rspec` tests for Puppet.

Files and paths on Windows

Puppet and Windows handle directory separators and line endings in files somewhat differently, so you must be aware of the differences when you are writing manifests to manage Windows systems.

Directory separators in file paths

Several resource types (including `file`, `exec`, and `package`) take file paths as values for various attributes. The Puppet language uses the backslash (`\`) as an escape character in quoted strings. However, Windows also uses the backslash to separate directories in file paths, such as `C:\Program Files\PuppetLabs`. Additionally, Windows file system APIs accept both backslashes and forward slashes in file paths, but some Windows programs accept only backslashes.

Generally, if Puppet itself is interpreting the file path, or if the file path is meant for the primary server, use forward slashes. If the file path is being passed directly to a Windows program, use backslashes. The following table lists common directory path uses and what kind of slashes are required by each.

File path usage	Slash type
Template paths, such as <code>template('my_module/content.erb')</code> .	Forward slash (/)
<code>puppet:///</code> URLs.	Forward slash (/)
The <code>path</code> attribute or title of a <code>file</code> resource.	Forward slash (/) or backslash (\)
The <code>source</code> attribute of a <code>package</code> resource.	Forward slash (/) or backslash (\)
Local paths in a <code>file</code> resource's <code>source</code> attribute.	Forward slash (/) or backslash (\)
The <code>command</code> of an <code>exec</code> resource. However, some executables, such as <code>cmd.exe</code> , require backslashes.	Forward slash (/) or backslash (\)
Any file paths included in the <code>command</code> of a <code>scheduled_task</code> resource.	Backslash (\)
Any file paths included in the <code>install_options</code> of a <code>package</code> resource.	Backslash (\)

File path usage	Slash type
Any file paths used for Windows PowerShell DSC resources. For these resources, single quote strings whenever possible.	Backslash (\)

Line endings in files

Windows uses CRLF line endings instead of *nix's LF line endings. Be aware of the following issues:

- If you specify the contents of a file with the `content` attribute, Puppet writes the content in binary mode. To create files with CRLF line endings, specify the `\r\n` escape sequence as part of the content.
- When downloading a file to a Windows node with the `source` attribute, Puppet transfers the file in binary mode, leaving the original newlines untouched.
- If you are using version control, such as Git, ensure that it is configured to use CRLF line endings.
- Non-file resource types that make partial edits to a system file, such as the [host resource type](#), which manages the `%windir%\system32\drivers\etc\hosts` file, manage their files in text mode and automatically translate between Windows and *nix line endings.

Note: When writing your own resource types, you can get this behavior by using the `flat` file type.

Related information

[Templates](#) on page 553

Templates are written in a specialized templating language that generates text from data. Use templates to manage the content of your Puppet configuration files via the `content` attribute of the `file` resource type.

[Strings](#) on page 542

Strings are unstructured text fragments of any length. They're a common and useful data type.

Code comments

To add comments to your Puppet code, use shell-style or hash comments.

Hash comments begin with a hash symbol (`#`) and continue to the end of a line. You can start comments either at the beginning of a line or partway through a line that began with code.

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

Variables

Variables store values so that those values can be accessed in code later.

After you've assigned a variable a value, you cannot reassign it. Variables depend on order of evaluation: you must assign a variable a value before it can be resolved.

The following video gives you an overview of variables:

Note: Puppet contains built-in variables that you can use in your manifests. For a list of these, see the page on [facts and built-in variables](#).

Assigning variables

Prefix variable names with a dollar sign (`$`). Assign values to variables with the equal sign (`=`) assignment operator.

Variables accept values of any data type. You can assign literal values, or you can assign any statement that resolves to a normal value, including expressions, functions, and other variables. The variable then contains the value that the statement resolves to, rather than a reference to the statement.

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

Assign variables using their short name within their own scope. You cannot assign values in one scope from another scope. For example, you can assign a value to the `apache::params` class's `$vhostdir` variable only from within the `apache::params` class.

You can assign multiple variables at the same time from an array or hash.

To assign multiple variables from an array, you must specify an equal number of variables and values. If the number of variables and values do not match, the operation fails. You can also use nested arrays. For example, all of the variable assignments shown below are valid.

```
[ $a, $b, $c ] = [ 1, 2, 3 ]      # $a = 1, $b = 2, $c = 3
[ $a, [ $b, $c ] ] = [ 1, [ 2, 3 ] ] # $a = 1, $b = 2, $c = 3
[ $a, $b ] = [ 1, [ 2 ] ]         # $a = 1, $b = [ 2 ]
[ $a, [ $b ] ] = [ 1, [ 2 ] ]     # $a = 1, $b = 2
```

When you assign multiple variables with a hash, list the variables in an array on the left side of the assignment operator, and list the hash on the right. Hash keys must match their corresponding variable name. This example assigns a value of 10 to the `$a` variable and a value of 20 to the `$b` variable.

```
[ $a, $b ] = { a => 10, b => 20 } # $a = 10, $b = 20
```

You can include extra key-value pairs in the hash, but all variables to the left of the operator must have a corresponding key in the hash:

```
[ $a, $c ] = { a => 5, b => 10, c => 15, d => 22 } # $a = 5, $c = 15
```

Puppet allows a given variable to be assigned a value only one time within a given scope. This is a little different from most programming languages. You cannot change the value of a variable, but you can assign a different value to the same variable name in a new scope:

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

Resolution

You can use the name of a variable in any place where a value of the variable's data type would be accepted, including expressions, functions, and resource attributes. Puppet replaces the name of the variable with its value. By default, unassigned variables have a value of `undef`. See the section about unassigned variables and strict mode for more details.

In these examples, the `content` parameter value resolves to whatever value has been assigned to the `$content` variable. The `$address_array` variable resolves to an array of the values assigned to the `$address1`, `$address2`, and `$address3` variables:

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

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

Interpolation

Puppet can resolve variables that are included in double-quoted strings; this is called *interpolation*. Inside a double-quoted string, surround the name of the variable (the portion after the `$`) with curly braces, such as `${var_name}`. This syntax is optional, but it helps to avoid ambiguity and allows variables to be placed directly next to non-whitespace characters. These optional curly braces are permitted only inside strings.

For example, the curly braces make it easier to quickly identify the variable `$homedir`.

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

Scope

The area of code where a given variable is visible is dictated by its *scope*. Variables in a given scope are available only within that scope and its child scopes, and any local scope can locally override the variables it receives from its parents. See the section on [scope](#) for complete details.

You can access out-of-scope variables from named scopes by using their qualified names, which include namespaces representing the variable's scope. The scope is where the variable is defined and assigned a value. For example, the qualified name of this `$vhost` variable shows that the variable is found and assigned a value in the `apache::params` class:

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

Variables can be assigned outside of any class, type, or node definition. These *top scope* variables have an empty string as their first namespace segment, so that the qualified name of a top scope variable begins with a double colon, such as `$::osfamily`.

Unassigned variables and strict mode

By default, you can access variables that have never had values assigned to them. If you do, their value is `undef`. This can be a problem, because an unassigned variable is often an accident or a typo. To make unassigned variable usage return an error, so that you can find and fix the problem, enable strict mode by setting `strict_variables = true` in the `puppet.conf` file on your primary server and on any nodes that run puppet apply. For details about this setting, see the [configuration](#) page.

Related information

[Expressions and operators](#) on page 422

Expressions are statements that resolve to values. You can use expressions almost anywhere a value is required. Expressions can be compounded with other expressions, and the entire combined expression resolves to a single value.

[Function calls](#) on page 438

Functions are plug-ins, written in Ruby, that you can call during catalog compilation. A call to any function is an expression that resolves to a value. Most functions accept one or more values as arguments, and return a resulting value.

Naming variables

Some variable names are reserved; for detailed information, see the [reserved name](#) page.

Variable names

Variable names are case-sensitive and must begin with a dollar sign (\$). Most variable names must start with a lowercase letter or an underscore. The exception is regex capture variables, which are named with only numbers.

Variable names can include:

- Uppercase and lowercase letters
- Numbers
- Underscores (_). If the first character is an underscore, access that variable only from its own local scope.

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

Optionally, the name of the very first namespace can be empty, representing the top namespace. The main reason to namespace this way is to indicate to anyone reading your code that you're accessing a top-scope variable, such as `$::is_virtual`.

You can also use a regular expression for variable names. Short variable names match the following regular expression:

```
\A\[a-z0-9_][a-zA-Z0-9_]*\Z
```

Qualified variable names match the following regular expression:

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

Resources

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

The following video gives you an overview of resources:

Resources contained in classes and defined types share the relationships of those classes and defined types. Resources are not subject to scope: a resource in any area of code can be referenced from any other area of code.

A *resource declaration* adds a resource to the catalog and tells Puppet to manage that resource's state.

When Puppet applies the compiled catalog, it:

1. Reads the actual state of the resource on the target system.
2. Compares the actual state to the desired state.
3. If necessary, changes the system to enforce the desired state.
4. Logs any changes made to the resource. These changes appear in Puppet agent's log and in the run report, which is sent to the primary server and forwarded to any specified report processors.

If the catalog doesn't contain a particular resource, Puppet does nothing with whatever that resource described. If you remove a package resource from your manifests, Puppet doesn't uninstall the package; instead, it just ignores it. To remove a package, manage it as a resource and set `ensure => absent`.

You can delay adding resources to the catalog. For example, classes and defined types can contain groups of resources. These resources are managed only if you add that class or defined resource to the catalog. Virtual resources are added to the catalog only after they are realized.

Resource declarations

At minimum, every resource declaration has a *resource type*, a *title*, and a set of *attributes*:

```
<TYPE> { '<TITLE>' : <ATTRIBUTE> => <VALUE>, }
```

The resource title and attributes are called the resource body. A resource declaration can have one resource body or multiple resource bodies of the same resource type.

Resource declarations are expressions in the Puppet language — they always have a side effect of adding a resource to the catalog, but they also resolve to a value. The value of a resource declaration is an array of *resource references*, with one reference for each resource the expression describes.

A resource declaration has extremely low precedence; in fact, it's even lower than the variable assignment operator (`=`). This means that if you use a resource declaration for its value, you must surround it with parentheses to associate it with the expression that uses the value.

If a resource declaration includes more than one resource body, it declares multiple resources of that resource type. The resource body is a title and a set of attributes; each body must be separated from the next one with a semicolon. Each resource in a declaration is almost completely independent of the others, and they can have completely different values for their attributes. The only connections between resources that share an expression are:

- They all have the same resource type.
- They can all draw from the same pool of default values, if a resource body with the title `default` is present.

Resource uniqueness

Each resource must be unique; Puppet does not allow you to declare the same resource twice. This is to prevent multiple conflicting values from being declared for the same attribute. Puppet uses the resource `title` and the `name` attribute or *namevar* to identify duplicate resources — if either the title or the name is duplicated within a given resource type, catalog compilation fails. See the page about [resource syntax](#) for details about resource titles and namevars. To provide the same resource for multiple classes, use a class or a virtual resource to add it to the catalog in multiple places without duplicating it. See [classes](#) and [virtual resources](#) for more information.

Relationships and ordering

By default, Puppet applies unrelated resources in the order in which they're written in the manifest. If a resource must be applied before or after some other resource, declare a relationship between them to show that their order isn't coincidental. You can also make changes in one resource cause a refresh of some other resource. See the [Relationships and ordering](#) page for more information.

Otherwise, you can customize the default order in which Puppet applies resources with the `ordering` setting. See the [configuration page](#) for details about this setting.

Resource types

Every resource is associated with a *resource type*, which determines the kind of configuration it manages. Puppet has built-in resource types such as `file`, `service`, and `package`. See the [resource type reference](#) for a complete list and information about the built-in resource types.

You can also add new resource types to Puppet:

- Defined types are lightweight resource types written in the Puppet language.

- Custom resource types are written in Ruby and have the same capabilities as Puppet's built-in types.

Title

A resource's title is a string that uniquely identifies the resource to Puppet. In a resource declaration, the title is the identifier after the first curly brace and before the colon. For example, in this file resource declaration, the title is `/etc/passwd`:

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

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

The title of a resource differs from the *namevar* of the resource. Whereas the title identifies the resource to Puppet itself, the namevar identifies the resource to the target system and is usually specified by the resource's *name* attribute. The resource title doesn't have to match the namevar, but you'll often want it to: the value of the namevar attribute defaults to the title, so using the name in the title can save you some typing.

If a resource type has multiple namevars, the type specifies whether and how the title maps to those namevars. For example, the `package` type uses the `provider` attribute to help determine uniqueness, but that attribute has no special relationship with the title. See each type's documentation for details about how it maps title to namevars.

Attributes

Attributes describe the desired state of the resource; each attribute handles some aspect of the resource. For example, the `file` type has a `mode` attribute that specifies the permissions for the file.

Each resource type has its own set of available attributes; see the [resource type reference](#) for a complete list. Most resource types have a handful of crucial attributes and a larger number of optional ones. Attributes accept certain data types, such as strings, numbers, hashes, or arrays. Each attribute that you declare must have a value. Most attributes are optional, which means they have a default value, so you do not have to assign a value. If an attribute has no default, it is considered required, and you must assign it a value.

Most resource types contain an `ensure` attribute. This attribute generally manages the most basic state of the resource on the target system, such as whether a file exists, whether a service is running or stopped, or whether a package is installed or uninstalled. The values accepted for the `ensure` attribute vary by resource type. Most accept `present` and `absent`, but there are variations. Check the reference for each resource type you are working with.

Tip: Resource and type attributes are sometimes referred to as parameters. Puppet also has properties, which are slightly different from parameters: properties correspond to something measurable on the target system, whereas parameters change how Puppet manages a resource. A property always represents a concrete state on the target system. When talking about resource declarations in Puppet, parameter is a synonym for attribute.

Namevars and `name`

Every resource on a target system must have a unique identity; you cannot have two services, for example, with the same name. This identifying attribute in Puppet is known as the *namevar*.

Each resource type has an attribute that is designated to serve as the namevar. For most resource types, this is the `name` attribute, but some types use other attributes, such as the `file` type, which uses `path`, the file's location on disk, for its namevar. If a type's namevar is an attribute other than `name`, this is listed in the type reference documentation.

Most types have only one namevar. With a single namevar, the value must be unique per resource type. There are a few rare exceptions to this rule, such as the `exec` type, where the namevar is a command. However, some resource types, such as `package`, have multiple namevar attributes that create a composite namevar. For example, both the `yum` provider and the `gem` provider have `mysql` packages, so both the `name` and the `provider` attributes are namevars, and Puppet uses both to identify the resource.

The `namevar` differs from the resource's *title*, which identifies a resource to Puppet's compiler rather than to the target system. In practice, however, a resource's `namevar` and the title are often the same, because the `namevar` usually defaults to the title. If you don't specify a value for a resource's `namevar` when you declare the resource, Puppet uses the resource's title.

You might want to specify different a `namevar` that is different from the title when you want a consistently titled resource to manage something that has different names on different platforms. For example, the NTP service might be `ntpd` on Red Hat systems, but `ntp` on Debian and Ubuntu. You might title the service "ntp," but set its `namevar` --- the `name` attribute --- according to the operating system. Other resources can then form relationships to the resource without the title changing.

Metaparameters

Some attributes in Puppet can be used with every resource type. These are called *metaparameters*. These don't map directly to system state. Instead, metaparameters affect Puppet's behavior, usually specifying the way in which resources relate to each other.

The most commonly used metaparameters are for specifying order relationships between resources. See the documentation on [relationships and ordering](#) for details about those metaparameters. See the full list of available metaparameters in the [metaparameter reference](#).

Resource syntax

You can accomplish a lot with just a few resource declaration features, or you can create more complex declarations that do more.

Basic syntax

The simplified form of a resource declaration includes:

- The resource type, which is a word with no quotes.
- An opening curly brace {.
- The title, which is a string.
- A colon (:).
- Optionally, any number of attribute and value pairs, each of which consists of:
 - An attribute name, which is a lowercase word with no quotes.
 - A `=>` (called an arrow, "fat comma," or "hash rocket").
 - A value, which can have any [data type][datatype].
 - A trailing comma.
- A closing curly brace }.

You can use any amount of whitespace in the Puppet language.

This example declares a file resource with the title `/etc/passwd`. This declaration's `ensure` attribute ensures that the specified file is created, if it does not already exist on the node. The rest of the declaration sets values for the file's `owner`, `group`, and `mode` attributes.

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

Complete syntax

By creating more complex resource declarations, you can:

- Describe many resources at once.
- Set a group of attributes from a hash with the `*` attribute.

- Set default attributes.
- Specify an abstract resource type.
- Amend or override attributes after a resource is already declared.

The complete generalized form of a resource declaration expression is:

- The resource type, which can be one of:
 - A lowercase word with no quotes, such as `file`.
 - A resource type data type, such as `File`, `Resource[File]`, or `Resource['file']`. It must have a type but not a title.
- An opening curly brace (`{`).
- One or more resource bodies, separated with semicolons (`;`). Each resource body consists of:
 - A title, which can be one of:
 - A string.
 - An array of strings, which declares multiple resources.
 - The special value `default`, which sets default attribute values for other resource bodies in the same expression.
 - A colon (`:`).
 - Optionally, any number of attribute and value pairs, separated with commas (`,`). Each attribute/value pair consists of:
 - An attribute name, which can be one of:
 - A lowercase word with no quotes.
 - The special attribute `*`, called a "splat," which takes a hash and sets other attributes.
 - A `=>`, called an arrow, a "fat comma," or a "hash rocket".
 - A value, which can have any data type.
 - Optionally, a trailing comma after the last attribute/value pair.
 - Optionally, a trailing semicolon after the last resource body.
- A closing curly brace (`}`)

```
<TYPE> { default: * => <HASH OF ATTRIBUTE/VALUE PAIRS>, <ATTRIBUTE> =>
  <VALUE>, ; '<TITLE>': * => <HASH OF ATTRIBUTE/VALUE PAIRS>, <ATTRIBUTE> =>
  <VALUE>, ; '<NEXT TITLE>': ... ; ['<TITLE>', '<TITLE>', '<TITLE>']: ... ;
```

Resource declaration default attributes

If a resource declaration includes a resource body with a title of `default`, Puppet doesn't create a new resource named "default." Instead, every other resource in that declaration uses attribute values from the `default` body if it doesn't have an explicit value for one of those attributes. This is also known as "per-expression defaults."

Resource declaration defaults are useful because it lets you set many attributes at once, but you can still override some of them.

This example declares several different files, all using the default values set in the `default` resource body. However, the `mode` value for the files in the last array (`['ssh_config', 'ssh_host_dsa_key.pub' ...]`) is set explicitly instead of using the default.

```
file {
  default:
    ensure => file,
    owner  => "root",
    group  => "wheel",
    mode   => "0600",
  ;
  ['ssh_host_dsa_key', 'ssh_host_key', 'ssh_host_rsa_key']:
    # use all defaults
```

```

;
['ssh_config', 'ssh_host_dsa_key.pub', 'ssh_host_key.pub',
'ssh_host_rsa_key.pub', 'sshd_config']:
  # override mode
  mode => "0644",
;
}

```

The position of the default body in a resource declaration doesn't matter; resources above and below it all use the default attributes if applicable. You can only have one default resource body per resource declaration.

Setting attributes from a hash

You can set attributes for a resource by using the splat attribute, which uses the splat or asterisk character `*`, in the resource body.

The value of the splat (`*`) attribute must be a hash where:

- Each key is the name of a valid attribute for that resource type, as a string.
- Each value is a valid value for the attribute it's assigned to.

This sets values for that resource's attributes, using every attribute and value listed in the hash.

For example, the splat attribute in this declaration sets the `owner`, `group`, and `mode` settings for the file resource.

```

$file_ownership = {
  "owner" => "root",
  "group" => "wheel",
  "mode"  => "0644",
}

file { ["/etc/passwd":
  ensure => file,
  *      => $file_ownership,
]
}

```

You cannot set any attribute more than once for a given resource; if you try, Puppet raises a compilation error. This means that:

- If you use a hash to set attributes for a resource, you cannot set a different, explicit value for any of those attributes. For example, if `mode` is present in the hash, you can't also set `mode => "0644"` in that resource body.
- You can't use the `*` attribute multiple times in one resource body, since the splat itself is an attribute.

To use some attributes from a hash and override others, either use a hash to set per-expression defaults, as described in the section on resource declaration defaults, or use the merging operator, `+` to combine attributes from two hashes, with the right-hand hash overriding the left-hand one.

Abstract resource types

Because a resource declaration can accept a resource type data type as its resource type, you can use a `Resource[<TYPE>]` value to specify a non-literal resource type, where the `<TYPE>` portion can be read from a variable. That is, the following three examples are equivalent to each other:

```

file { ["/tmp/foo": ensure => file, ] File { ["/tmp/foo": ensure => file, ]
Resource[File] { ["/tmp/foo": ensure => file, ]

```

```

$mytype = File
Resource[$mytype] { ["/tmp/foo": ensure => file, ]

```

```
$mytypename = "file"
Resource[$mytypename] { ["/tmp/foo": ensure => file, }
```

This lets you declare resources without knowing in advance what type of resources they'll be, which can enable transformations of data into resources.

Arrays of titles

If you specify an array of strings as the title of a resource body, Puppet creates multiple resources with the same set of attributes. This is useful when you have many resources that are nearly identical.

For example:

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

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

If you do this, you must let the namevar attributes of these resources default to their titles. You can't specify an explicit value for the namevar, because it applies to all of the resources.

Adding or modifying attributes

Although you cannot declare the same resource twice, you can add attributes to an resource that has already been declared. In certain circumstances, you can also override attributes. You can amend attributes with either a resource reference, a collector, or from a hash using the splat (*) attribute.

To amend attributes with the splat attribute, see the section about setting attributes from a hash.

To amend attributes with a resource reference, add a resource reference attribute block to the resource that's already declared. Normally, you can only use resource reference blocks to add previously unmanaged attributes to a resource; it cannot override already-specified attributes. The general form of a resource reference attribute block is:

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

For example, this resource reference attribute block amends values for the `owner`, `group`, and `mode` attributes:

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

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

You can also amend attributes with a collector.

The general form of a collector attribute block is:

- A [resource collector](#) that matches any number of resources
- An opening curly brace
- Any number of attribute => value (or attribute +> value) pairs
- A closing curly brace

For resource attributes that accept multiple values in an array, such as the relationship metaparameters, you can add to the existing values instead of replacing them by using the "plusignment" (+>) keyword instead of the usual hash rocket (=>). For details, see appending to attributes in the [classes](#) documentation.

This example amends the owner, group, and mode attributes of any resources that match the collector:

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

include base::linux

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



CAUTION: Be very careful when amending attributes with a collector. Test with `--noop` to see what changes your code would make.

- It can override other attributes you've already specified, regardless of class inheritance.
- It can affect large numbers of resources at one time.
- It implicitly realizes any virtual resources the collector matches.
- Because it ignores class inheritance, it can override the same attribute more than one time, which results in an evaluation order race where the last override wins.

Local resource defaults

Because resource default statements are subject to dynamic scope, you can't always tell what areas of code will be affected. Generally, do not include classic resource default statements anywhere other than in your site manifest (`site.pp`). See the [resource defaults documentation](#) for details. Whenever possible, use resource declaration defaults, also known as per-expression defaults.

However, resource default statements can be powerful, allowing you to set important defaults, such as file permissions, across resources. Setting local resource defaults is a way to protect your classes and defined types from accidentally inheriting defaults from classic resource default statements.

To set local resource defaults, define your defaults in a variable and re-use them in multiple places, by combining resource declaration defaults and setting attributes from a hash.

This example defines defaults in a `$file_defaults` variable, and then includes the variable in a resource declaration default with a hash.

```
class mymodule::params {
  $file_defaults = {
    mode   => "0644",
    owner  => "root",
    group  => "root",
  }
  # ...
}
```

```
class mymodule inherits mymodule::params {
  file { default: * => $mymodule::params::file_defaults;
    "/etc/myconfig":
      ensure => file,
  };
}
```

Resource types

Every resource (file, user, service, package, and so on) is associated with a resource type within the Puppet language. The resource type defines the kind of configuration it manages. This section provides information about the resource types that are built into Puppet.

- [All resource types](#)
- [Core types cheat sheet](#) on page 391

This page provides a reference guide for the core Puppet types: package, file, service, notify, exec, cron, user, and group.

- [Optional resource types for Windows](#) on page 397

In addition to the resource types included with Puppet, you can install custom resource types as modules from the Forge. This is especially useful when managing Windows systems, because there are several important Windows-specific resource types that are developed as modules rather than as part of core Puppet.

- [exec](#)
- [Using exec on Windows](#) on page 397

Puppet uses the same `exec` resource type on both *nix and Windows systems, and there are a few Windows-specific best practices and tips to keep in mind.

- [file](#)
- [Using file on Windows](#) on page 398

Use Puppet's built-in `file` resource type to manage files and directories on Windows, including ownership, group, permissions, and content, with the following Windows-specific notes and tips.

- [filebucket](#)
- [group](#)
- [Using user and group on Windows](#) on page 400

Use the built-in `user` and `group` resource types to manage user and group accounts on Windows.

- [index](#)
- [notify](#)
- [package](#)
- [Using package on Windows](#) on page 401

The built-in `package` resource type handles many different packaging systems on many operating systems, so not all features are relevant everywhere. This page offers guidance and tips for working with `package` on Windows.

- [resources](#)
- [schedule](#)
- [service](#)
- [Using service](#) on page 403

Puppet can manage services on nearly all operating systems. This page offers operating system-specific advice and best practices for working with `service`.

- [stage](#)
- [tidy](#)
- [user](#)

Core types cheat sheet

This page provides a reference guide for the core Puppet types: package, file, service, notify, exec, cron, user, and group.

For detailed information about these types, see the [Resource type reference](#) or the other pages in this section.

The trifecta: package, file, and service

Package, file, service: Learn it, live it, love it. Even if this is the only Puppet you know, you can get a lot done.


```
package { 'openssh-server':
  ensure => installed,
}

file { ['/etc/ssh/sshd_config':
  source  => 'puppet:///modules/sshd/sshd_config',
  owner   => 'root',
  group   => 'root',
  mode    => '0640',
  notify  => Service['sshd'], # sshd restarts whenever you edit this file.
  require => Package['openssh-server'],
}

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

package

Manages software packages.

Attribute	Description	Notes
name	The name of the package, as known to your packaging system.	Defaults to title.
ensure	Whether the package should be installed, and what version to use.	<p>Allowed values:</p> <ul style="list-style-type: none"> • present • latest (implies present) • Any version string (implies present) • absent • purged <p> CAUTION: purged ensures absent, and deletes configuration files and dependencies, including those that other packages depend on. Provider-dependent.</p>
source	Where to obtain the package, if your system's packaging tools don't use a repository.	
provider	Which packaging system to use (such as Yum or Rubygems), if a system has more than one available.	

file

Manages files, directories, and symlinks.

Attribute	Description	Notes
ensure	Whether the file should exist, and what it should be.	Allowed values: <ul style="list-style-type: none"> • file • directory • link (symlink) • present (anything) • absent
path	The full path to the file on disk.	Defaults to title.
owner	By name or UID.	
group	By name or GID.	
mode	Must be specified exactly. Does the right thing for directories.	

For normal files:

source	Where to download contents for the file. Usually a puppet:/// URL.
content	The file's desired contents, as a string. Most useful when paired with templates , but you can also use the output of the <code>file</code> function.

For directories:

source	Where to download contents for the directory, when <code>recurse => true</code> .
recurse	Whether to recursively manage files in the directory.
purge	Whether unmanaged files in the directory should be deleted, when <code>recurse => true</code> .

For symlinks:

target	The symlink target. (Required when <code>ensure => link</code> .)
--------	--

Other notable attributes:

- backup
- checksum
- force
- ignore
- links
- recurselimit
- replace

service

Manages services running on the node. As with packages, some platforms have better tools than others, so read the relevant documentation before you begin.

You can make services restart whenever a file changes with the `subscribe` or `notify` metaparameters. For more info, see [Relationships and ordering](#).

Attribute	Description	Notes
<code>name</code>	The name of the service to run.	Defaults to title.
<code>ensure</code>	The desired status of the service.	Allowed values: <ul style="list-style-type: none"> <code>running</code> (or <code>true</code>) <code>stopped</code> (or <code>false</code>)
<code>enable</code>	Whether the service should start on boot. Doesn't work on all systems.	
<code>hasrestart</code>	Whether to use the init script's restart command instead of <code>stop+start</code> .	Defaults to false.
<code>hasstatus</code>	Whether to use the init script's status command.	Defaults to true.

Other notable attributes:

If a service has a bad init script, you can work around it and manage almost anything using the `status`, `start`, `stop`, `restart`, `pattern`, and `binary` attributes.

Other core types

Beyond package, file, and service, these core types are among the most useful and commonly used.

notify

Logs an arbitrary message, at the `notice` log level. This appears in the POSIX syslog or Windows Event Log on the agent node and is also logged in reports.

```
notify { "This message is getting logged on the agent node.": }
```

Attribute	Description	Notes
<code>message</code>	The message to log.	Defaults to title.

exec

Executes an arbitrary command on the agent node. When using execs, you must either make sure the command can be safely run multiple times, or specify that it runs only under certain conditions.

Important attributes	Description	Notes
<code>command</code>	The command to run. If this isn't a fully-qualified path, use the <code>path</code> attribute.	Defaults to title.
<code>path</code>	Where to look for executables, as a colon-separated list or an array.	
<code>returns</code>	Which exit codes indicate success.	Defaults to 0.

Important attributes	Description	Notes
environment	An array of environment variables to set (for example, ['MYVAR=somevalue' , 'OTHERVAR=othervalue']).	
The following attributes limit when a command runs.		
creates	A file to look for before running the command. The command only runs if the file doesn't exist.	
refreshonly	If true, the command runs only if a resource it subscribes to (or a resource which notifies it) has changed.	
onlyif	A command or array of commands; if any have a non-zero return value, the command won't run.	
unless	The opposite of onlyif.	

Other notable attributes: cwd, group, logoutput, timeout, tries, try_sleep, user

cron

Manages cron jobs. On Windows, use `scheduled_task` instead.

```
cron { 'logrotate':
  command => "/usr/sbin/logrotate",
  user    => "root",
  hour    => 2,
  minute  => 0,
}
```

Important attributes	Description	Notes
command	The command to execute.	
ensure	Whether the job should exist.	Allowed values: <ul style="list-style-type: none"> present absent
hour, minute, month, monthday, weekday	The timing of the cron job.	

Other notable attributes: environment, name, special, target, user

user

Manages user accounts; mostly used for system users.

```
user { "jane":
  ensure => present,
  uid    => '507',
  gid    => 'admin',
}
```

```

shell      => '/bin/zsh',
home       => '/home/jane',
managehome => true,
}

```

Important Attributes	Description	Notes
name	The name of the user.	Defaults to title.
ensure	Whether the user should exist.	Allowed values: <ul style="list-style-type: none"> • present • absent • role
uid	The user ID. Must be specified numerically; chosen automatically if omitted.	Read-only on Windows.
gid	The user's primary group. Can be specified numerically or by name.	Not used on Windows; use <code>groups</code> instead.
groups	An array of other groups to which the user belongs.	Don't include the group specified as the <code>gid</code> .
home	The user's home directory.	
managehome	Whether to manage the home directory when managing the user.	If you don't set this to <code>true</code> , you'll need to create the user's home directory manually.
shell	The user's login shell.	

Other notable attributes: `comment`, `expiry`, `membership`, `password`, `password_max_age`, `password_min_age`, `purge_ssh_keys`, `salt`

group

Manages groups.

Important attributes	Description	Notes
name	The name of the group.	Defaults to title.
ensure	Whether the group should exist.	Allowed values: <ul style="list-style-type: none"> • present • absent
gid	The group ID; must be specified numerically, and is chosen automatically if omitted.	Read-only on Windows.
members	Users and groups that are members of the group.	Only applicable to certain operating systems; see the full type reference for details.

Optional resource types for Windows

In addition to the resource types included with Puppet, you can install custom resource types as modules from the Forge. This is especially useful when managing Windows systems, because there are several important Windows-specific resource types that are developed as modules rather than as part of core Puppet.

If you're doing heavy management of Windows systems, the following modules might be helpful:

- [puppetlabs/acl](#): A resource type for managing access control lists (ACLs) on Windows.
- [puppetlabs/registry](#): A resource type for managing arbitrary registry keys.
- [puppetlabs/reboot](#): A resource type for managing conditional reboots, which can be necessary for installing certain software.
- [puppetlabs/dism](#): A resource type for enabling and disabling Windows features (on Windows 7 or 2008 R2 and newer).
- [puppetlabs/powershell](#): An alternative `exec` provider that can directly execute PowerShell commands.

Other resource types created by community members are also available on the Forge. The best way to find new resource types is by searching for “Windows” on the Forge and exploring the results.

Remember: Plugins from the Forge might not have the same amount of quality assurance and test coverage as the core resource types included in Puppet.

Using `exec` on Windows

Puppet uses the same `exec` resource type on both *nix and Windows systems, and there are a few Windows-specific best practices and tips to keep in mind.

Puppet can run binary files (such as `exe`, `com`, or `bat`), and can log the child process output and exit status. To ensure the resource is idempotent, specify one of the `creates`, `onlyif`, or `unless` attributes.

Command extensions

If a file extension for the command is not specified (for example, `ruby` instead of `ruby.exe`), Puppet will use the `PATHEXT` environment variable to resolve the appropriate binary. `PATHEXT` is a Windows-specific variable that lists the valid file extensions for executables.

Exit codes

On Windows, most exit codes are integers between 0 and 2147483647.

Larger exit codes on Windows behave inconsistently across different tools. The Win32 APIs define exit codes as 32-bit unsigned integers, but both the `cmd.exe` shell and the .NET runtime cast them to signed integers. This means some tools will report negative numbers for exit codes above 2147483647. For example, `cmd.exe` reports 4294967295 as -1.

Because Puppet uses the `GetExitCodeProcess` Win32 API, it reports the very large number instead of the negative number, which might not be what you expect if you got the exit code from a `cmd.exe` session.

Microsoft recommends against using negative or very large exit codes, so avoid them.

Tip: To convert a negative exit code to the positive one Puppet will use, subtract it from 4294967296.

Shell built-ins

Puppet does not support a shell provider for Windows, so if you want to execute shell built-ins (such as `echo`), you must provide a complete `cmd.exe` invocation as the command. For example, `command => 'cmd.exe /c echo "hello"'`.

When using `cmd.exe` and specifying a file path in the command line, be sure to use backslashes. For example, `'cmd.exe /c type c:\path\to\file.txt'`. If you use forward slashes, `cmd.exe` returns an error.

Optional PowerShell `exec` provider

An optional PowerShell `exec` provider is available as a plug-in and is helpful if you need to run PowerShell commands from within Puppet. To use it, install [puppetlabs/powershell](#).

Inline PowerShell scripts

If you choose to execute PowerShell scripts using the default Puppet `exec` provider on Windows, you must specify the `remotesigned` execution policy as part of the `powershell.exe` invocation:

```
exec { 'test':
  command => 'C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -
  executionpolicy remotesigned -file C:\test.ps1',
}
```

Using `file` on Windows

Use Puppet's built-in `file` resource type to manage files and directories on Windows, including ownership, group, permissions, and content, with the following Windows-specific notes and tips.

```
file { 'c:/mysql/my.ini':
  ensure => 'file',
  mode   => '0660',
  owner  => 'mysql',
  group  => 'Administrators',
  source => 'N:/software/mysql/my.ini',
}
```

Take care with backslashes in file paths

The issue of backslashes and forward-slashes in file paths can get complicated. See [Handling file paths on Windows](#) for more information.

Be consistent with capitalization in file names

If you refer to a file resource in multiple places in a manifest (such as when creating [relationships between resources](#)), be consistent with the capitalization of the file name. If you use `my.ini` in one place, don't use `MY.INI` in another place.

Windows NTFS filesystems are case-insensitive (albeit case-preserving); Puppet is case-sensitive. Windows itself won't be confused by inconsistent case, but Puppet will think you're referring to different files.

Make sure the Puppet user account has appropriate permissions

To manage files properly, Puppet needs the following Windows privileges:

- Create symbolic links
- Back up files and directories
- Restore files and directories

When Puppet runs as a service, make sure its user account is a member of the local `Administrators` group. When you use the [PUPPET_AGENT_ACCOUNT_USER](#) parameter with the MSI installer, the user will automatically be added to the `Administrators` group.

Before running Puppet interactively (on Windows Vista or 2008 and later versions), start the command prompt window with elevated privileges by right-clicking on the start menu and choosing "Run as Administrator."

Managing file permissions: the `mode` attribute and the `acl` module

The permissions models used by *nix and Windows are quite different. When you use [the `mode` attribute](#), the `file` type manages them both like *nix permissions, and translates the mode to roughly

equivalent access controls on Windows. This makes basic controls fairly simple, but doesn't work for managing complex access rules.

If you need fine-grained Windows access controls, use the `puppetlabs/acl` module, which provides an optional `acl` resource type that manages permissions in a Windows-centric way. Leave mode unspecified and add an `acl` resource. See [the `acl` module's documentation](#) for details.

How *nix modes map to Windows permissions

*nix permissions are expressed as either a quoted octal number (such as `"755"`), or a string of symbolic modes, (such as `"u=rwx,g=rx,o=rx"`). See [the reference for the `file` type's `mode` attribute](#) for more details about the syntax.

These mode expressions generally manage three kinds of permission — read, write, execute — for three kinds of user — owner, group, other. They translate to Windows permissions as follows:

- The read, write, and execute permissions are interpreted as the `FILE_GENERIC_READ`, `FILE_GENERIC_WRITE`, and `FILE_GENERIC_EXECUTE` access rights, respectively.
- The `Everyone` SID is used to represent users other than the owner and group.
- Directories on Windows can have the sticky bit, which makes it so users can delete files only if they own the containing directory.
- The owner of a file can be a group (for example, `owner => 'Administrators'`) and the group of a file can be a user (for example, `group => 'Administrator'`).
- While it's possible for the owner and group to be the same, this is strongly discouraged. Doing so can cause problems when the mode gives different permissions to the owner and group (such as `0750`).
- The group can't have higher permissions than the owner. Other users can't have higher permissions than the owner or group. In other words, `0640` and `0755` are supported, but `0460` is not.

Extra behavior when managing permissions with `mode`

When you manage permissions with the `mode` attribute, it has the following side effects:

- The owner of a file or directory always has the `FULL_CONTROL` access right.
- The security descriptor is always set to protected. This prevents the file from inheriting more permissive access controls from the directory that contains it.

File sources

The `source` attribute of a file can be a Puppet URL, a local path, a UNC path, or a path to a file on a mapped drive.

Handling line endings

Windows usually uses CRLF line endings, rather than the LF line endings used by *nix. In most cases, Puppet **does not** automatically convert line endings when managing files on Windows.

If a file resource uses the `content` or `source` attributes, Puppet writes the file in binary mode, using the line endings that are present in the content. If the manifest, template, or source file is saved with CRLF line endings, Puppet uses those endings in the destination file.

Non-file resource types that make partial edits to a system file (most notably the `host` resource type, which manages the `%windir%\system32\drivers\etc\hosts` file) manage their files in text mode, and automatically translate between Windows and *nix line endings.

Note: When writing your own resource types, you can get this same behavior by using the `flat` file type.

Using `user` and `group` on Windows

Use the built-in `user` and `group` resource types to manage user and group accounts on Windows.

Managing local user and group resources

Puppet uses the `user` and `group` resource types to manage local accounts. You can't write a Puppet resource that describes a domain user or group. However, a local group resource can manage which domain accounts belong to the local group.

Managing group membership with Puppet

Windows manages group membership by specifying the groups to which a user belongs, or by specifying the members of a group. Puppet supports both of these methods.

When Puppet is managing a local user, you can list the groups that the user belongs to. These groups can be a local group account (such as `Administrators`) or a domain group account.

When Puppet is managing a local group, you can list the members that belong to the group. Each member can be a local account (such as `Administrator`) or a domain account, where each account can be a user or a group account.

When managing a user, Puppet makes sure that the user belongs to all of the groups listed in the manifest. If the user belongs to a group not specified in the manifest, Puppet does not remove the user from the group.

If you want to ensure that a user belongs to only the groups listed in the manifest, and no others, specify the `membership` attribute for the user. If set to `inclusive`, Puppet removes the user from any group not listed in the manifest.

Similarly, when managing a group, Puppet makes sure all of the members listed in the manifest are added to the group. Existing members of the group who are not listed in the manifest are ignored.

To ensure that a group contains only the members listed in the manifest, and no others, specify the `auth_membership` attribute for the group. When this attribute is present and set to `true`, Puppet removes any members of the group not listed in the manifest.

Allowed `user` attributes on Windows

When managing Windows user accounts, you can use the following `user` resource type attributes:

Attribute	Usage notes
<code>name</code>	
<code>ensure</code>	
<code>comment</code>	
<code>groups</code>	You cannot use the <code>gid</code> attribute with Windows.
<code>home</code>	
<code>managehome</code>	
<code>membership</code>	
<code>password</code>	Passwords must be specified in cleartext, because Windows does not have an API for setting the password hash.
<code>auth_membership</code>	
<code>uid</code>	Read-only. Available for inspecting a user by running <code>puppet resource user <NAME></code> . The <code>uid</code> value will be the user's SID (see below).

Allowed group attributes on Windows

When managing Windows group accounts, you can use the following [group](#) resource type attributes:

Attribute	Usage notes
name	
ensure	
members	
auth_membership	
gid	Read-only. Available for inspecting a group by running <code>puppet resource group <NAME></code> . The <code>gid</code> value will be the group's SID (see below).

Names and security identifiers (SIDs)

On Windows, user and group account names can take multiple forms, such as:

- Administrators
- <host>\Administrators
- BUILTIN\Administrators
- S-1-5-32-544

The S-1-5-32-544 name form is called a security identifier (SID). Puppet treats all these forms equally: when comparing two account names, it transforms account names into their canonical SID form and compares the SIDs.

When you refer to a user or group in multiple places in a manifest (such as when creating [relationships between resources](#)), be consistent with how you capitalize the name. Names are case-sensitive in Puppet manifests, but case-insensitive on Windows. It's important that the cases match, however, because autorequire will attempt to match users with fully qualified names (such as `User[BUILTIN\Administrators]`) in addition to SIDs (such as `User[S-1-5-32-544]`). It might not match in cases where domain accounts and local accounts have the same name, such as `Domain\Bob` versus `LOCAL\Bob`.

Note: When listed for reporting or by `puppet resource`, groups always return the fully qualified form when describing a user, such as `BUILTIN\Administrators`. These fully qualified names might not look the same as in the names specified in the manifest.

Using package on Windows

The built-in package resource type handles many different packaging systems on many operating systems, so not all features are relevant everywhere. This page offers guidance and tips for working with package on Windows.

```
package { 'mysql':
  ensure      => '5.5.16',
  source      => 'N:\packages\mysql-5.5.16-winx64.msi',
  install_options => ['INSTALLDIR=C:\mysql-5.5'],
}

package { "Git version 1.8.4-preview20130916":
  ensure      => installed,
  source      => 'C:\code\puppetlabs\temp\windowsexample\Git-1.8.4-
  preview20130916.exe',
  install_options => ['/VERYSILENT']
}
```

Supported package types: MSI and EXE

Puppet can install and remove MSI packages and executable installers on Windows. Both package types use the default windows package provider.

Alternatively, a [Chocolatey package provider](#) is available on the Forge.

The `source` attribute is required

When managing packages using the windows package provider, you must specify a package file using the `source` attribute.

The source can be a local file or a file on a mapped network drive. For MSI installers, you can use a UNC path. Puppet URLs are not supported for the package type's `source` attribute, but you can use `file` resources to copy packages to the local system. The `source` attribute accepts both forward- and backslashes.

The package name must be the `DisplayName`

The title (or name) of the package must match the value of the package's `DisplayName` property in the registry, which is also the value displayed in the **Add/Remove Programs** or **Programs and Features** control panels.

If the provided name and the installed name don't match, Puppet assumes the package is not installed and tries to install it again.

To determine a package's `DisplayName`:

1. Install the package on an example system.
2. Run `puppet resource package` to see a list of installed packages.
3. Copy the name of the package from the list.

Some packages (Git is a notable example) change their display names with every newly released version. See the section below on handling package versions and upgrades.

Install and uninstall options

The Windows package provider also supports package-specific `install_options` (such as install directory) and `uninstall_options`. These options vary across packages, so see the documentation for the specific package you're installing. Options are specified as an array of strings or hashes.

MSI properties can be specified as an array of strings following the `property=key` pattern; use one string per property. Command line flags to executable installers can be specified as an array of strings, with one string per flag.

For file path arguments within the `install_options` attribute (such as `INSTALLDIR`), use backslashes (`\`), not forward slashes. Escape your backslashes appropriately. For more info, see [Handling file paths on Windows](#).

Use the hash notation for file path arguments because they might contain spaces. For example:

```
install_options => [ { 'INSTALLDIR' => ${packagedir} } ]
```

Handling versions and upgrades

Setting `ensure => latest` (which requires the `upgradeable` feature) doesn't work on Windows, because Windows doesn't have central package repositories like on most *nix systems.

There are two ways to handle package versions and upgrades on Windows.

Packages with real versions

Many packages on Windows have version metadata. To tell whether a package has usable version metadata, install it on a test system and use `puppet resource package` to inspect it.

To upgrade these packages, replace the `source` file and set `ensure => '<VERSION>'`. For example:

```
package { 'mysql':
  ensure      => '5.5.16',
  source      => 'N:\packages\mysql-5.5.16-winx64.msi',
  install_options => ['INSTALLDIR=C:\mysql-5.5'],
}
```

The next time Puppet runs, it will notice that the versions don't match and will install the new package. This makes the installed version match the new version, so Puppet won't attempt to re-install the package until you change the version in the manifest again.

The version you use in `ensure` must exactly match the version string that the package reports when you inspect it with `puppet resource`. If it doesn't match, Puppet will repeatedly try to install it.

Packages that include version info in their `DisplayName`

Some packages don't embed version metadata; instead, they change their `DisplayName` property with each release. The Git packages are a notable example.

To upgrade these packages, replace the `source` file and update the resource's title or name to the new `DisplayName`. For example:

```
package { "Git version 1.8.4-preview20130916":
  ensure      => installed,
  source      => 'C:\code\puppetlabs\temp\windowsexample\Git-1.8.4-
  preview20130916.exe',
  install_options => ['/VERYSILENT']
}
```

The next time Puppet runs, it will notice that the names don't match and will install the new package. This makes the installed name match the new name, so Puppet won't attempt to re-install the package until you change the name in the manifest again.

The name you use in the title must exactly match the name that the package reports when you inspect it with `puppet resource`. If it doesn't match, Puppet will repeatedly try to install it.

Using `service`

Puppet can manage services on nearly all operating systems. This page offers operating system-specific advice and best practices for working with `service`.

Using `service` on *nix systems

If your *nix operating system has a good system for managing services, and all the services you care about have working init scripts or service configs, you can write small `service` resources with just the `ensure` and `enable` attributes.

For example:

```
service { 'apache2':
  ensure => running,
  enable => true,
}
```

Note: Some *nix operating systems don't support the `enable` attribute.

Defective init script

On platforms that use SysV-style init scripts, Puppet assumes the script has working `start`, `stop`, and `status` commands. See descriptions below.

If the `status` command is missing, set `hasstatus => false` for that service. This makes Puppet search the process table for the service's name to check whether it's running.

In some rare cases — such as virtual services like the Red Hat `network` — a service won't have a matching entry in the process table. If a service acts like this and is also missing a `status` command, set `hasstatus => false` and also specify either `status` or `pattern` attribute.

No init script or service config

If some of your services lack init scripts, Puppet can compensate, as in the following example:

```
service { "apache2":
  ensure => running,
  start  => "/usr/sbin/apachectl start",
  stop   => "/usr/sbin/apachectl stop",
  pattern => "/usr/sbin/httpd",
}
```

In addition to specifying `ensure`, specify also how to start the service, how to stop it, how to check whether it's running, and optionally how to restart it.

Start

Use either `start` or `binary` to specify a start command. The difference is that `binary` also gives you default behavior for `stop` and `status`.

Stop

If you specified `binary`, Puppet defaults to finding that same executable in the process table and killing it.

To stop the service some other way, use the `stop` attribute to specify the appropriate command.

Status

If you specified `binary`, Puppet checks for that executable in the process table. If it doesn't find it, it reports that the service isn't running.

If there's a better way to check the service's status, or if the `start` command is just a script and a different process implements the service itself, use either `status` (a command that exits 0 if the service is running and nonzero otherwise) or `pattern` (a pattern to search the process table for).

Restart

If a service needs to be reloaded, Puppet defaults to stopping it and starting it again. If you have a safer command for restarting a service, you can optionally specify it in the `restart` attribute.

Using service on macOS

macOS handles services much like most *nix-based systems. The main difference is that `enable` and `ensure` are much more closely linked — running services are always enabled, and stopped ones are always disabled.

For best results, either leave `enable` blank or make sure it's set to `true` whenever `ensure => running`.

The `launchd` plists that configure your services must be in one of the following directories:

- `/System/Library/LaunchDaemons`
- `/System/Library/LaunchAgents`
- `/Library/LaunchDaemons`
- `/Library/LaunchAgents`

You can also specify `start` and `stop` commands to assemble your own services, much like on *nix systems.

Using service on Windows

On Windows, Puppet can start, stop, enable, disable, list, query, and configure services. It expects that all services run with the built-in Service Control Manager (SCM) system. It does not support configuring service dependencies, the account to run as, or desktop interaction.

When writing `service` resources for Windows, remember the following:

- Use the short service name (such as `wuauserv`) in Puppet, not the display name (such as `Automatic Updates`).
- Set `enable => true` to assign a service the Automatic startup type.
- Set `enable => manual` to assign the Manual startup type.

For example, here is a complete service resource:

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

Puppet 7 adds support for managing the logon user and password to the Windows service provider. For example:

```
service { 'name-of-service':
  ensure      => 'running',
  enable      => 'true',
  logonaccount => 'domain\\user',
  logonpassword => $password,
}
```

Note that the `logonpassword` is a sensitive variable.

Managing systemd services

In addition to the default values for the `ensure` and `enable` attributes, you can mask systemd services by setting `enable => mask`.

Note that static services can not be enabled by design, and services of type `indirect` can not be enabled due to a limitation in the implementation of systemd. When restarting services, Puppet checks whether any unit files were modified and runs `daemon-reload` when required.

Relationships and ordering

Resources are included and applied in the order they are defined in their manifest, but only if the resource has no implicit relationship with another resource, as this can affect the declared order. To manage a group of resources in a specific order, explicitly declare such relationships with relationship metaparameters, chaining arrows, and the `require` function.

To override Puppet's default manifest ordering, declare an explicit relationship between resources. All relationships cause Puppet to manage specific resources before other resources. Relationships are not limited by evaluation-order; you can declare a relationship with a resource before that resource has been declared.

Refreshing and notification

Some resource types can refresh when one of their dependencies changes. For example, some services must restart when their configuration files change, so `service` resources can refresh by restarting the service.

The built-in resource types that can refresh are `service`, `exec`, and `package`. For specific details about these types, see the [resource reference](#).

To specify that a resource must refresh when a related resource changes, create a notifying relationship with the `subscribe` or `notify` metaparameters or the notification chaining arrow (`~>`). When a resource changes, it sends a refresh event to any resources that subscribe to it. Those resources that are subscribed receive the refresh event.

When receiving refresh events:

- If a resource gets a refresh event during a run, and its resource type has a refresh action, it performs that action.
- If a resource gets a refresh event, but its resource type cannot refresh, nothing happens.
- If a class or defined resource gets a refresh event, every resource it contains also gets a refresh event.
- A resource can perform its refresh action up to once per run. If it receives multiple refresh events, they're combined, and the resource refreshes only once.

When sending refresh events:

- If a resource is not in its desired state, and Puppet makes changes to it during a run, it sends a refresh event to any subscribed resources.
- If a resource performs its refresh action during a run, it sends a refresh event to any subscribed resources.

- If Puppet changes or refreshes any resource in a class or defined resource, that class or defined resource sends a refresh event to any subscribed resources.

If non-operational (no-op) mode is enabled:

- The resource does not refresh when it receives a refresh event. Instead, Puppet logs a message stating what would have happened.
- The resource does not send refresh events to subscribed resources. Instead, Puppet logs messages stating what would have happened to any resources further down the subscription chain.

For more information about refresh behavior, see the types documentation.

Automatic relationships

Certain resource types can have automatic relationships with other resources, using `autorequire`, `autonotify`, `autobefore`, or `autosubscribe`. This creates an ordering relationship without you explicitly stating one.

Puppet establishes automatic relationships between types and resources when it applies a catalog. It searches the catalog for any resources that match certain rules and processes them in the correct order, sending refresh events if necessary. If any explicit relationship, such as those created by chaining arrows, conflicts with an automatic relationship, the explicit relationship take precedence.

Missing dependencies

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

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

Failed dependencies

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

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

It then continues to apply any unrelated resources. Any resources that depend on the skipped resource are also skipped. This helps prevent an inconsistent system state, rather than attempting to apply a resource that might have broken prerequisites.

Dependency cycles

If two or more resources require each other in a loop, Puppet compiles the catalog but won't be able to apply it. Puppet logs an error like the following, and attempts to help identify the cycle:

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

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

Related information

[Containment](#) on page 584

Containment is what controls the order in which the various parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types, determining what has to happen before other things can happen.

Relationship metaparameters

You can use certain metaparameters to establish relationships by setting any of them as an attribute in any resource.

The following video gives you an overview of metaparameters:

Set the value of any relationship metaparameter to either a resource reference or an array of references that point to one or more target resources:

- **before:** Applies a resource before the target resource.
- **require:** Applies a resource after the target resource.
- **notify:** Applies a resource before the target resource. The target resource refreshes if the notifying resource changes.
- **subscribe:** Applies a resource after the target resource. The subscribing resource refreshes if the target resource changes.

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

The two examples below create the same ordering relationship, ensuring that the `openssh-server` package is managed before the `sshd_config` file:

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

```
file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
  require => Package['openssh-server'],
}
```

The two examples below create the same notifying relationship, so that if Puppet changes the `sshd_config` file, it sends a notification to the `sshd` service:

```
file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
  notify => Service['sshd'],
}
```

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

Because an array of resource references can contain resources of differing types, these two examples also create the same ordering relationship. In both examples, Puppet manages the `openssh-server` package and the `sshd_config` file before it manages the `sshd` service.

```
service { 'sshd':
  ensure => running,
  require => [
    Package['openssh-server'],
    File['/etc/ssh/sshd_config'],
  ],
}
```

```
package { 'openssh-server':
  ensure => present,
  before => Service['sshd'],
}

file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
  before => Service['sshd'],
}
```

Related information

[Resources](#) on page 383

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

[Resource and class references](#) on page 535

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

[The Array data type](#) on page 518

The data type of arrays is `Array`. By default, `Array` matches arrays of any length, provided all values in the array match the abstract data type `Data`. You can use parameters to restrict which values `Array` matches.

Chaining arrows

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

The ordering arrow is a hyphen and a greater-than sign (`->`). It applies the resource on the left before the resource on the right.

The notifying arrow is a tilde and a greater-than sign (`~>`). It applies the resource on the left first. If the left-hand resource changes, the right-hand resource refreshes.

In this example, Puppet applies configuration to the `ntp.conf` file resource and notifies the `ntpd` service resource if there are any changes.

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

Note: When possible, use relationship metaparameters, not chaining arrows. Metaparameters are more explicit and easier to maintain. See the Puppet language [style guide](#) for information on when and how to use chaining arrows.

Operands

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

- Resource references, including multi-resource references.
- Arrays of resource references.
- Resource declarations.
- Resource collectors.

You can link operands to apply a series of relationships and notifications. In this example, Puppet applies configuration to the package, notifies the file resource if there are changes, and then, if there are resulting changes to the file resource, Puppet notifies the service resource:

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

Resource declarations can be chained. That means you can use chaining arrows to make Puppet apply a section of code in the order that it is written. This example applies configuration to the package, the file, and the service, in that order, with each related resource notifying the next of any changes:

```
# first:
package { 'openssh-server':
  ensure => present,
} # and then:
-> file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => '0600',
  source => 'puppet:///modules/sshd/sshd_config',
} # and then:
~> service { 'sshd':
  ensure => running,
  enable => true,
}
```

Collectors can also be chained, so you can create relationships between many resources at one time. This example applies all Yum repository resources before applying any package resources, which protects any packages that rely on custom repositories:

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

Capturing resource references for generated resources

In Puppet, the value of a resource declaration is a reference to the resource it creates.

This is useful if you're automatically creating resources whose titles you can't predict: use the iteration functions to declare several resources at once or use an array of strings as a resource title. If you assign the resulting resource references to a variable, you can then use them in chaining statements without ever knowing the final title of the affected resources.

For example:

- The `map` function iterates over its arguments and returns an array of values, with each value produced by the last expression in the block. If that last expression is a resource declaration, `map` produces an array of resource references, which you could then use as an operand for a chaining arrow.
- For a resource declaration whose title is an array, the value is itself an array of resource references that you can assign to a variable and use in a chaining statement.

Cautions when chaining resource collectors

Chains can create dependency cycles.

Chained collectors can cause huge dependency cycles; be careful when using them. They can also be dangerous when used with virtual resources, which are implicitly realized by collectors.

Chains can break.

Although you can usually chain many resources or collectors together (`File['one'] -> File['two'] -> File['three']`), the chain can break if it includes a collector whose search expression doesn't match any resources.

Implicit properties aren't searchable.

Collectors can search only on attributes present in the manifests; they cannot see properties that are automatically set or are read from the target system. For example, the chain `Yumrepo <| |> -> Package <| provider == yum |>`, creates only relationships with packages whose `provider` attribute is explicitly set to `yum` in the manifests. It would not affect packages that didn't specify a provider but use `Yum` because it's the operating system's default provider.

Reversed forms

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

Related information

[Resource and class references](#) on page 535

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

[Resources](#) on page 383

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

[Resource collectors](#) on page 575

Resource collectors select a group of resources by searching the attributes of each resource in the catalog, even resources which haven't yet been declared at the time the collector is written. Collectors realize virtual resources, are used in chaining statements, and override resource attributes. Collectors have an irregular syntax that enables them to function as a statement and a value.

[Lambdas](#) on page 572

Lambdas are blocks of Puppet code passed to functions. When a function receives a lambda, it provides values for the lambda's parameters and evaluates its code. If you use other programming languages, think of lambdas as anonymous functions that are passed to other functions.

[Virtual resources](#) on page 577

A virtual resource declaration specifies a desired state for a resource without enforcing that state. Puppet manages the resource by realizing it elsewhere in your manifests. This divides the work done by a normal resource declaration into two steps. Although virtual resources are declared one time, they can be realized any number of times, similar to a class.

The require function

Use the `require` function to declare a class and make it a dependency of the surrounding container.

For example:

```
class wordpress {
  require apache
  require mysql
  ...
}
```

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

Unlike the relationship metaparameters and chaining arrows, the `require` function does not have a reciprocal form or a notifying form. However, you can create more complex behavior by combining `include` and chaining arrows inside a class definition. This example notifies and restarts every service in the `apache::ssl` class if any of the SSL certificates on the node change:

```
class apache::ssl {
  include site::certificates
  Class['site::certificates'] ~> Class['apache::ssl']
}
```

Classes

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

The following video gives you an overview of classes:

Defining classes

Defining a class makes it available for later use. It doesn't add any resources to the catalog — to do that, you must declare the class or assign it from an external node classifier (ENC).

Create a class by writing a class definition in a manifest (`.pp`) file. Store class manifests in the `manifests/` directory of a module. Define only one class in a manifest, and give the manifest file the same name as the class. Puppet automatically loads any classes that are present in a valid module. See [module fundamentals](#) to learn more about module structure and usage.

A class contains all of its resources. This means any relationships formed with the class as a whole is extended to every resource in the class. Every resource in a class gets automatically tagged with the class's name and each of its namespace segments.

Classes can contain other classes, but you must use the `contain` function to explicitly specify when a class is contained. For more information, see the documentation about [containing classes](#). A contained class is automatically tagged with the name of its container.

Tip: Unlike many parts of Puppet code, class definitions aren't expressions, so you can't use them where a value is expected.

The general form of a class definition is:

- The `class` keyword.
- The name of the class.
- An optional parameter list, which consists of:
 - An opening parenthesis.
 - A comma-separated list of parameters, such as `String $myparam = "value"`. Each parameter consists of:
 - An optional data type, which restricts the allowed values for the parameter. If not specified, the data type defaults to `Any`.
 - A variable name to represent the parameter, including the dollar sign (\$) prefix
 - An optional equals sign (=) and default value, which must match the data type, if one was specified.
 - An optional trailing comma after the last parameter.
 - A closing parenthesis.
- Optionally, the `inherits` keyword followed by a single class name.
- An opening curly brace.
- A block of arbitrary Puppet code, which generally contains at least one resource declaration.

- A closing curly brace.

For example, this class definition specifies no parameters:

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

This class definition creates a version parameter (`$version`) that accepts a String data type with a default value of 'latest'. It also includes file content from an embedded Ruby (ERB) template from the `apache` module.

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

Class parameters and variables

Parameters allow a class to request external data. If a class needs to use data other than facts for configuration, use a parameter for that data.

You can use class parameters as normal variables inside the class definition. The values of these variables are set based on user input when the class is declared, rather than with normal assignment statements.

Supply default values for parameters whenever possible. If a class parameter lacks a default value, the parameter is considered required and the user must set a value, either in external data or as an override.

If you set a data type for each parameter, Puppet checks the parameter's value at runtime to make sure that it is the correct data type, and raises an error if the value is illegal. If you do not provide a data type for a parameter, the parameter accepts values of any data type.

The variables `$title` and `$name` are both set to the class name automatically, so you can't use them as parameters.

Setting class parameter defaults with Hiera data

To set class parameter defaults with Hiera data in your modules, set up a hierarchy in your module's `hiera.yaml` file and include the referenced data files in the data directory.

For example, this `hiera.yaml` file, located in the root directory of the `ntp` module, uses the operating system fact to determine which class defaults to apply to the target system. Puppet first looks for a data file that matches the

operating system of the target system: `path: "os/{facts.os.family}.yaml"`. If no matching path is found, Puppet uses defaults from the "common" data file instead.

```
# ntp/hiera.yaml
---
version: 5
defaults:
  datadir: data
  data_hash: yaml_data
hierarchy:
  - name: "OS family"
    path: "os/{facts.os.family}.yaml"

  - name: "common"
    path: "common.yaml"
```

The files in the example below specify the default values are located in the data directory:

- `Debian.yaml` specifies the defaults for systems that return an operating system fact of Debian.
- `common.yaml` specifies the defaults for all other systems.

```
# ntp/data/common.yaml
---
ntp::autoupdate: false
ntp::service_name: ntpd

# ntp/data/os/Debian.yaml
ntp::service_name: ntp
```

Tip:

If you are maintaining older modules, you might encounter cases where class parameter defaults are set with a parameter class, such as `params.pp`, and class inheritance. Update such modules to use Hiera data instead. Class inheritance can have unpredictable effects and makes troubleshooting difficult. For details about updating existing `params` classes to Hiera data, see [data in modules](#).

Related information

[Variables](#) on page 380

Variables store values so that those values can be accessed in code later.

[Resources](#) on page 383

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

[Tags](#) on page 581

Tags are useful for collecting resources, analyzing reports, and restricting catalog runs. Resources, classes, and defined type instances can have multiple tags associated with them, and they receive some tags automatically.

[Scope](#) on page 586

A scope is a specific area of code that is partially isolated from other areas of code.

[Values, data types, and aliases](#) on page 504

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

[Namespaces and autoloading](#) on page 590

Class and defined type names can be broken up into segments called namespaces which enable the autoloader to find the class or defined type in your modules.

Declaring classes

Declaring a class in a Puppet manifest adds all of its resources to the catalog.

You can declare classes in node definitions, at top scope in the site manifest, and in other classes or defined types. Classes are singletons — although a given class can behave very differently depending on how its parameters are set, the resources in it are evaluated only once per compilation. You can also assign classes to nodes with an external node classifier (ENC) .

Puppet has two main ways to declare classes: include-like and resource-like. Include-like declarations are the most common; they are flexible and idempotent, so you can safely repeat them without causing errors. Resource-like declarations are mostly useful if you want to pass parameters to the class but can't or don't use Hiera. Most ENCs assign classes with include-like behavior, but others assign them with resource-like behavior. See the [ENC interface](#) documentation or the documentation of your specific ENC for details.



CAUTION: Do not mix include-like and resource-like declarations for a given class. If you declare or assign a class using both styles, it can cause compilation failures.

Include-like declarations

Include-like resource declarations allow you to declare a class multiple times — but no matter how many times you add the class, it is added to the catalog only once. This allows classes or defined types to manage their own dependencies and allows you create overlapping *role* classes, in which a given node can have more than one role.

Include-like behavior relies on external data and defaults for class parameter values, which allows the external data source to act like cascading configuration files for all of your classes.

You can declare a class with this behavior with one of four functions: `include`, `require`, `contain`, and `hiera_include`.

When a class is declared with an include-like declaration, Puppet takes the following actions, in order, for each of the class parameters:

1. Requests a value from the external data source, using the key `<class name>::<parameter name>`. For example, to get the `apache` class's `version` parameter, Puppet searches for `apache::version`.
2. Uses the default value, if one exists.
3. Fails compilation with an error, if no value is found.

The `include` function

The `include` function is the most common way to declare classes. Declaring a class with this function includes the class in the catalog.

Tip: The `include` function refers only to inclusion in the catalog. You can include a class in another class's definition, but doing so does not mean one class contains the other; it only means the included class will be added to the catalog. If you want one class to contain another, use the `contain` function instead.

This function uses include-like behavior, so you can make multiple declarations and Puppet relies on external data for parameters.

The `include` function accepts one of the following:

- A single class name, such as `apache`.
- A single class reference, such as `Class['apache']`.
- A comma-separated list of class names or class references.
- An array of class names or class references.

This single class name declaration declares the class only once and has no additional effect:

```
include base::linux
```

This example declares a single class with a class reference:

```
include Class[ 'base::linux' ]
```

This example declares two classes in a list:

```
include base::linux, apache
```

This example declares two classes in an array:

```
$my_classes = [ 'base::linux', 'apache' ]
include $my_classes
```

The `require` function

The `require` function declares one or more classes, then causes them to become a dependency of the surrounding container. This function uses include-like behavior, so you can make multiple declarations, and Puppet relies on external data for parameters.

Tip: The `require` function is used to declare classes and defined types. Do not confuse it with the `require` metaparameter, which is used to establish relationships between resources.

The `require` function accepts one of the following:

- A single class name, such as `apache`.
- A single class reference, such as `Class['apache']`.
- A comma-separated list of class names or class references.
- An array of class names or class references.

In this example, Puppet ensures that every resource in the `apache` class is applied before any resource in any `apache::vhost` instance:

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

The `contain` function

The `contain` function is used inside another class definition to declare one or more classes and *contain* those classes in the surrounding class. This enforces ordering of classes. When you contain a class in another class, the relationships of the containing class extend to the contained class as well. For details about containment, see the documentation on [containing classes](#).

This function uses include-like behavior, so you can make multiple declarations, and Puppet relies on external data for parameters.

The `contain` function accepts one of the following:

- A single class name, such as `apache`.
- A single class reference, such as `Class['apache']`.
- A comma-separated list of class names or class references.
- An array of class names or class references.

In this example class declaration, the `ntp` class contains the `ntp::service` class. Any resource that forms a relationship with the `ntp` class also has the same relationship to the `ntp::service` class.

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

For example, if a resource has a `before` relationship with the `ntp` class, that resource will also be applied before the `ntp::service` class. Similarly, any resource that forms a `require` relationship with `ntp` will be applied after `ntp::service`.

The `hiera_include` function

The `hiera_include` function requests a list of class names from Hiera, then declares all of them.

This function uses include-like behavior, so you can make multiple declarations, and Puppet relies on external data for parameters. The `hiera_contain` function accepts a single lookup key.

Because `hiera_include` uses the array lookup type, it gets a combined list that includes classes from every level of the hierarchy. This allows you to abandon node definitions and use Hiera like a lightweight external node classifier. For more information, see the [Hiera documentation](#).

For example, this `hiera_include` declaration in the site manifest applies classes across the site infrastructure, as specified in Hiera.

```
# /etc/puppetlabs/code/environments/production/manifests/site.pp
hiera_include(classes)
```

Given the Hiera data below, the node `web01.example.com` in the production environment gets the classes `apache`, `memcached`, `wordpress`, and `base::linux`. On all other nodes, only the `base::linux` class is declared.

```
# /etc/puppetlabs/puppet/hiera.yaml
...
hierarchy:
  - "%{::clientcert}"
  - common

# /etc/puppetlabs/code/hieradata/web01.example.com.yaml
---
classes:
  - apache
  - memcached
  - wordpress

# /etc/puppetlabs/code/hieradata/common.yaml
---
classes:
  - base::linux
```


Resource-like declarations

Resource-like class declarations require that you declare a given class only once. They allow you to override class parameters at compile time — for any parameters you don't override, Puppet falls back to external data.

Resource-like declarations must be unique to avoid conflicting parameter values. Repeated overrides cause catalog compilation to be unreliable and dependent on order evaluation. This is because overridden values from the class declaration:

- Always take precedence.
- Are computed at compile time.
- Do not have a built-in hierarchy for resolving conflicts.

When a class is declared with a resource-like declaration, Puppet takes the following actions, in order, for each of the class parameters:

1. Uses the override value from the declaration, if present.
2. Requests a value from the external data source, using the key `<class name>::<parameter name>`. For example, to get the `apache` class's `version` parameter, Puppet searches for `apache::version`.
3. Uses the default value.
4. Fails compilation with an error, if no value is found.

Resource-like declarations look like normal resource declarations, using the `class` pseudo-resource type. You can provide a value for any class parameter by specifying it as a resource attribute.

You can also specify a value for any metaparameter. In such cases, every resource contained in the class will also have that metaparameter. However:

- Any resource can specifically override metaparameter values received from its container.
- Metaparameters that can take more than one value, such as the `relationships` metaparameters, merge the values from the container and any resource-specific values.
- You cannot apply the `noop` metaparameter to resource-like class declarations.

For example, this resource-like declaration declares a class with no parameters:

```
class { 'base::linux': }
```

This declaration declares a class and specifies the version parameter:

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

Related information

[Node definitions](#) on page 441

A node definition, also known as a node statement, is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.

[Main manifest directory](#) on page 298

Puppet starts compiling a catalog either with a single manifest file or with a directory of manifests that are treated like a single file. This starting point is called the *main manifest* or *site manifest*.

[Defined resource types](#) on page 418

Defined resource types, sometimes called defined types or defines, are blocks of Puppet code that can be evaluated multiple times with different parameters.

[Relationships and ordering](#) on page 405

Resources are included and applied in the order they are defined in their manifest, but only if the resource has no implicit relationship with another resource, as this can affect the declared order. To manage a group of resources in a specific order, explicitly declare such relationships with relationship metaparameters, chaining arrows, and the `require` function.

[Containment](#) on page 584

Containment is what controls the order in which the various parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types, determining what has to happen before other things can happen.

[Resources](#) on page 383

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

[Classifying nodes](#) on page 329

You can classify nodes using an external node classifier (ENC), which is a script or application that tells Puppet which classes a node must have. It can replace or work in concert with the node definitions in the main site manifest (`site.pp`).

Defined resource types

Defined resource types, sometimes called defined types or defines, are blocks of Puppet code that can be evaluated multiple times with different parameters.

Create a defined resource type by writing a `define` statement in a manifest (`.pp`) file. You can declare a resource of a defined type in the same way you would declare a resource of a built-in type.

Store defined resource type manifests in the `manifests/` directory of a module. Define only one defined type in a manifest, and give the manifest file the same name as the defined type. Puppet automatically loads any defined types that are present in a valid module. See [module fundamentals](#) to learn more about module structure and usage.

If a defined type is present and loadable, you can declare resources of that defined type anywhere in your manifests. Declaring a new resource of the defined type causes Puppet to re-evaluate the block of code in the definition, using different values for the parameters.

Every instance of a defined type contains all of its unique resources. This means that any relationships formed between the instance and another resource are extended to every resource that makes up the instance. See the topics about [containment](#) and [relationships](#) for more information.

Tip: Unlike many parts of Puppet code, define statements aren't expressions, so you can't use them where a value is expected.

Defining types

The general form of a define statement is:

- The `define` keyword.
- The name of the defined type.
- An optional parameter list, which consists of:
 - An opening parenthesis.
 - A comma-separated list of parameters, such as: `String $myparam = "default value"`. Each parameter consists of:
 - An optional data type, which restricts the allowed values for the parameter. If no data type is specified, values of any data type are permitted.
 - A variable name to represent the parameter, including the `$` prefix, such as `$parameter`.
 - An optional `=` sign and default value, which must match the data type, if one was specified. If no default value is specified, the parameter is considered required and the user must specify a value.
 - An optional trailing comma after the last parameter.
 - A closing parenthesis.
- An opening curly brace.
- A block of arbitrary Puppet code, which generally contains at least one resource declaration
- A closing curly brace

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

This example creates a new resource type called `apache::vhost`:

```
# /etc/puppetlabs/puppet/modules/apache/manifests/vhost.pp
define apache::vhost (
  Integer $port,
  String[1] $docroot,
  String[1] $servername = $title,
  String $vhost_name = '*',
) {
  include apache # contains package['httpd'] and service['httpd']
  include apache::params # contains common config settings

  $vhost_dir = $apache::params::vhost_dir

  # the template used below can access all of the parameters and variable
  # from above.
  file { ["${vhost_dir}/${servername}.conf"]:
    ensure => file,
    owner  => 'www',
    group  => 'www',
    mode   => '0644',
    content => template('apache/vhost-default.conf.erb'),
    require => Package['httpd'],
    notify  => Service['httpd'],
  }
}
```

Declaring defined type resources

You can declare instances of a defined type—usually just called *resources*—the same way you declare any other resource: with a resource type, a title, and a set of attribute-value pairs. The parameters you added when defining the type, such as `$port`, become resource attributes, such as `port`, when you declare resources of the defined type.

Parameters that have a default value are considered optional parameters: if you don't specify them in the resource declaration, the default value is used. Parameters without defaults are required parameters, and you must specify a value for them when you declare the resource.

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

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

If a defined type is present and loadable, you can declare resources of that defined type anywhere in your manifests. Declaring a new resource of the defined type causes Puppet to re-evaluate the block of code in the definition, using the new declaration's values for the parameters.

Just as with a normal resource type, you can declare resource defaults for a defined type. In this example, every `apache::vhost` resource defaults to port 80 unless specifically overridden:

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

You can include any metaparameter in the declaration of a defined type instance. If you do:

- Every resource contained in the resource declaration also has that metaparameter. Metaparameters that can accept more than one value, such as the relationship metaparameters, merge the values from the container and any specific values from the individual resource.
- The value of the metaparameter can be used as a variable in the definition, as though it were a normal parameter. For example, in an instance declared with `require => Class['ntp']`, the local value of `$require` would be `Class['ntp']`.

Naming

Defined type names can consist of one or more *namespace* segments, which indicate the defined type's location in a module. Each segment must adhere to the [naming and reserved names](#) guidelines.

Each namespace segment must be capitalized when writing a resource reference, collector, or resource default. For example, a reference to the `apache::vhost` resource would be `Apache::Vhost['homepages']`.

Because you can declare multiple instances of a defined type in your manifests, every resource in the definition must be different in every instance. Duplicate resource instances result in compilation failures with a "duplicate resource declaration" error. To make resources different across instances, include the value of `$title` or another parameter in the resource's title and name.

Because `$title` is unique per instance, this ensures the resources are unique as well. For example, this segment of a file declaration makes resources unique by adding the `vhost_dir` and `servername` attributes to the resource title:

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

Parameters and attributes

When you create a defined type, you can precede each parameter in the define statement with an optional data type. If you include a data type, Puppet checks the resource parameter's value at runtime to make sure that it has the right data type; if the value is illegal, Puppet raises an error. If you don't specify a data type in the definition statement, the parameter accepts values of any data type.

You can use the parameters of a defined type as local variables inside the definition. Rather than the usual assignment statement, each instance of the defined type uses its parameter attributes to set the value of the variable. In this example declaration, the value of the `port` parameter, 8081, becomes the value assigned to the `$port` variable. Likewise, the path for the `docroot` parameter becomes the value for the `$docroot` variable.

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

Note:

The `$title` and `$name` variables are both set to the defined type's name automatically, so they cannot be used as parameters.

`$title` and `$name`

The `$title` and `$name` attributes are always available to a defined type and are not explicitly added to the definition. These attributes are both set to the defined type's name automatically:

- `$title` is always set to the title of the instance. Because it is always unique for each instance, it is useful for making sure that contained resources are unique.
- `$name` defaults to the value of `$title`. You can specify a different value when you declare an instance of the defined type, but this is rarely useful.

Because the values of `$title` and `$name` are already available inside the defined type's parameter list, you can use `$title` as all or part of the default value for another attribute. In this example, `$title` is used as the value of `$servername` to ensure the server name is always unique:

```
define apache::vhost (
  Integer $port,
  String[1] $docroot,
  String $servername = $title,
  String[1] $vhost_name = '*',
) { # ...
```

Related information

[Resources](#) on page 383

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

[Resource default statements](#) on page 575

Resource default statements enable you to set default attribute values for a given resource type. Resource declarations within the area of effect that omits those attributes inherit the default values.

[Containment](#) on page 584

Containment is what controls the order in which the various parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types, determining what has to happen before other things can happen.

[Namespaces and autoloading](#) on page 590

Class and defined type names can be broken up into segments called namespaces which enable the autoloader to find the class or defined type in your modules.

[Values, data types, and aliases](#) on page 504

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

Bolt tasks

Bolt tasks are single actions that you can run on target nodes in your infrastructure, allowing you to make as-needed changes to remote systems. You can run tasks with the Puppet Enterprise (PE) orchestrator or with Puppet's standalone task runner, Bolt.

Sometimes you need to do arbitrary tasks in your infrastructure that aren't about enforcing the state of machines. You might need to restart a service, run a troubleshooting script, or get a list of the network connections to a given node.

Tasks allow you to do actions like these with either the PE orchestrator or with Bolt, a standalone task runner. The orchestrator uses PE's built-in communication protocol, SSH, or WinRM to connect to the targets. Bolt connects to the targets with SSH or WinRM, without requiring any existing Puppet installation on the target. Bolt can also run plans, which chain multiple tasks together for more complex actions.

You can write tasks, which are a lot like scripts, in any programming language that can run on the target nodes, such as Bash, Python, or Ruby. Tasks are packaged within modules, so you can reuse, download, and share tasks on the Forge. Metadata for each task describes the task, validates input, and controls how the task runner executes the task.

For more information, see the [Bolt documentation on Tasks](#). If you're a Puppet Enterprise user, see [Running tasks](#) and [Using Bolt with orchestrator](#).

Expressions and operators

Expressions are statements that resolve to values. You can use expressions almost anywhere a value is required. Expressions can be compounded with other expressions, and the entire combined expression resolves to a single value.

In the Puppet language, nearly everything is an expression, including literal values, references to variables, resource declarations, function calls, and more. In other words, almost all statements in the language resolve to a value and can be used anywhere that value would be expected.

Most of this page is about expressions that are constructed with *operators*. Operators take input values and operate on them (for example, mathematically) to result in some other value. Other kinds of expressions (for example, function calls) are described in more detail on other pages.

Some expressions have side effects and are used in Puppet primarily for their side effects, rather than for their result value. For example:

- [Resource declaration](#): Adds a resource to the catalog.
- [Variable assignment](#): Creates a variable and assigns it a value.
- [Chaining statement](#): Forms a relationship between two or more resources.

Your code won't usually do anything with the value these expressions produce, but sometimes the value is useful for things like forming [relationships to resources whose names can't be predicted until run time](#).

Important: The following statements are not typical expressions. They don't resolve to usable values and can only be used in certain contexts:

- [Class definitions](#)
- [Defined types](#)
- [Node definitions](#)
- [Resource collectors](#)
- [Lambdas](#)

Expressions can be used almost everywhere, including:

- The operand of another expression.
- The condition of an if statement.
- The control expression of a case statement or selector statement.
- The assignment value of a variable.
- The argument or arguments of a function call.
- The title of a resource.
- An entry in an array
- A key or value of a hash.

Expressions cannot be used:

- Where a literal name of a class or defined type is expected (for example, in `class` or `define` statements).
- As the name of a variable (the name of the variable must be a literal name).
- Where a literal resource type or name of a resource type is expected (for example, in the type position of a resource declaration).

You can surround an expression by parentheses to control the order of evaluation in compound expressions (for example, `10+10/5` is 12, and `(10+10)/5` is 4), or to make your code clearer.

For formal descriptions of expressions constructed with operators and other elements of the Puppet language, see the Puppet [language specification](#).

Operator expressions

There are two kinds of operators:

- *Infix operators*, also called *binary operators*, appear between two operands:
 - `$a = 1`
 - `5 < 9`
 - `$operatingsystem != 'RedHat'`
- *Prefix operators*, also called *unary operators*, appear immediately before a single operand:
 - `*$interfaces`
 - `!$is_virtual`

Operands in an expression can be any other expression — anything that resolves to a value of the expected data type is allowed. Each operator has its own rules, described in the sections below, for the data types of its operands.

When you create compound expressions by using other expressions as operands, use parentheses for clarity and readability:

```
(90 < 7) and ('RedHat' == 'RedHat') # resolves to false
(90 < 7) or ('RedHat' in ['Linux', 'RedHat']) # resolves to true
```

Order of operations

Compound expressions are evaluated in a standard order of operations. Expressions wrapped in parentheses are evaluated first, starting from the innermost expression:

```
# This example resolves to 30, not 23:
notice( (7+8)*2 )
```

For the sake of clarity, use parentheses in all but the simplest compound expressions.

The precedence of operators, from highest to lowest, is:

Precedence	Operator
1	! (unary: not)
2	- (unary: numeric negation)
3	* (unary: array splat)
4	in
5	=~ and !~ (regex or data type match or non-match)
6	*, /, % (multiplication, division, and modulo)
7	+ and - (addition/array concatenation and subtraction/array deletion)
8	<< and >> (left shift and right shift)
9	== and != (equal and not equal)
10	>=, <=, >, and < (greater or equal, less or equal, greater than, and less than)
11	and
12	or
13	= (assignment)

Related information

[Values, data types, and aliases](#) on page 504

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

[Variables](#) on page 380

Variables store values so that those values can be accessed in code later.

[Resources](#) on page 383

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

[Function calls](#) on page 438

Functions are plug-ins, written in Ruby, that you can call during catalog compilation. A call to any function is an expression that resolves to a value. Most functions accept one or more values as arguments, and return a resulting value.

[Conditional statements and expressions](#) on page 431

Conditional statements let your Puppet code behave differently in different situations. They are most helpful when combined with facts or with data retrieved from an external source. Puppet supports *if* and *unless* statements, *case* statements, and *selectors*.

Comparison operators

Comparison operators take operands of several data types, and resolve to Boolean values.

Comparisons of numbers convert the operands to and from floating point and integer values, such that `1.0 == 1` is true. However, keep in mind that floating point values created by division are inexact, so mathematically equal values can be slightly unequal when turned into floating point values.

You can compare any two values with equals `==` or not equals `!=`, but only strings, numbers, and data types that require values to have a defined order can be compared with the less than or greater than operators.

Note: Comparisons of string values are case insensitive for characters in the US ASCII range. Characters outside this range are case sensitive.

Characters are compared based on their encoding. For characters in the US ASCII range, punctuation comes before digits, digits are in the order 0, 1, 2, ... 9, and letters are in alphabetical order. For characters outside US ASCII, ordering is defined by their UTF-8 character code, which might not always place them in alphabetical order for a given locale.

== (equality)

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

- Numbers: Tests simple equality.
- Strings: Tests whether two strings are identical, ignoring case as described in the Note, above.
- Arrays and hashes: Tests whether two arrays or hashes are identical.
- Booleans: Tests whether two Booleans are the same value.
- Data types: Tests whether two data types would match the exact same set of values.

Values are considered equal only if they have the same data type. Notably, this means that `1 == "1"` is false, and `"true" == true` is false.

!= (non-equality)

Resolves to `false` if the operands are equal. So, `$x != $y` is the same as `!($x == $y)`. It has the same behavior and restrictions, but opposite result, as equality `==`, above.

< (less than)

Resolves to `true` if the left operand is smaller than the right operand. Accepts numbers, strings, and data types; both operands must be the same type. When acting on data types, a less-than comparison is `true` if the left operand is a subset of the right operand.

> (greater than)

Resolves to `true` if the left operand is larger than the right operand. Accepts numbers, strings, and data types; both operands must be the same type. When acting on data types, a greater-than comparison is `true` if the left operand is a superset of the right operand.

<= (less than or equal to)

Resolves to `true` if the left operand is smaller than or equal to the right operand. Accepts numbers, strings, and data types; both operands must be the same type. When acting on data types, a less-than-or-equal-to comparison is `true` if the left operand is the same as the right operand or is a subset of it.

>= (greater than or equal to)

Resolves to `true` if the left operand is larger than or equal to the right operand. Accepts numbers, strings, and data types; both operands must be the same type. When acting on data types, a greater-than-or-equal-to comparison is `true` if the left operand is the same as the right operand or is a superset of it.

=~ (regex or data type match)

Resolves to `true` if the left operand matches the right operand. *Matching* means different things, depending on what the right operand is.

This operator is non-transitive with regard to data types. The right operand must be one of:

- A regular expression (regex), such as `/^[<>=]{7}/`.
- A stringified regular expression — that is, a string that represents a regular expression, such as `"^[<>=]{7}"`.
- A data type, such as `Integer[1,10]`.

If the right operand is a regular expression or a stringified regular expression, the left operand must be a string, and the expression resolves to `true` if the string matches the regular expression.

If the right operand is a data type, the left operand can be any value. The expression resolves to `true` if the left operand has the specified data type. For example, `5 =~ Integer` and `5 =~ Integer[1,10]` are both `true`.

!~ (regex or data type non-match)

Resolves to `false` if the left operand matches the right operand. So, `$x !~ $y` is the same as `!($x =~ $y)`. It has the same behavior and restrictions, but opposite result, as regex match `=~`, above.

in

Resolves to `true` if the right operand contains the left operand. The exact definition of "contains" here depends on the data type of the right operand. See table below.

This operator is non-transitive with regard to data types. It accepts:

- A string, regular expression, or data type as the left operand.
- A string, array, or hash as the right operand.

Expression	How in expression is evaluated
String in String	<p>Tests whether the left operand is a substring of the right, ignoring case:</p> <pre>'eat' in 'eaten' # resolves to true 'Eat' in 'eaten' # resolves to true</pre>
String in Array	<p>Tests whether one of the members of the array is identical to the left operand, ignoring case:</p> <pre>'eat' in ['eat', 'ate', 'eating'] # resolves to true 'Eat' in ['eat', 'ate', 'eating'] # resolves to true</pre>
String in Hash	<p>Tests whether the hash has a <i>key</i> identical to the left operand, ignoring case:</p> <pre>'eat' in { 'eat' => 'present tense', 'ate' => 'past tense' } # resolves to true 'eat' in { 'present' => 'eat', 'past' => 'ate' } # resolves to false</pre>
Regex in String	<p>Tests whether the right operand matches the regular expression:</p> <pre># note the case-insensitive option ? i /(?i:EAT)/ in 'eatery' # resolves to true</pre>
Regex in Array	<p>Tests whether one of the members of the array matches the regular expression:</p> <pre>/(?i:EAT)/ in ['eat', 'ate', 'eating'] # resolves to true</pre>

Expression	How in expression is evaluated
Regex in Hash	<p>Tests whether the hash has a <i>key</i> that matches the regular expression:</p> <pre>/(?i:EAT)/ in { 'eat' => 'present tense', 'ate' => 'past tense' } # resolves to true /(?i:EAT)/ in { 'present' => 'eat', 'past' => 'ate' } # resolves to false</pre>
Data type in Array	<p>Tests whether one of the members of the array matches the data type:</p> <pre># looking for integers between 100 and 199 Integer[100, 199] in [1, 2, 125] # resolves to true Integer[100, 199] in [1, 2, 25] # resolves to false</pre>
Data type in anything else	Always false.

Related information

[Booleans](#) on page 520

Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.

[Numbers](#) on page 530

Numbers in the Puppet language are normal integers and floating point numbers.

[Strings](#) on page 542

Strings are unstructured text fragments of any length. They're a common and useful data type.

[Arrays](#) on page 516

Arrays are ordered lists of values. Resource attributes which accept multiple values (including the relationship metaparameters) generally expect those values in an array. Many functions also take arrays, including the iteration functions.

[Hashes](#) on page 527

Hashes map keys to values, maintaining the order of the entries according to insertion order.

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Regular expressions](#) on page 533

A regular expression (sometimes shortened to “regex” or “regexp”) is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

Boolean operators

Boolean expressions resolve to boolean values. They are most useful when creating compound expressions.

A boolean operator takes boolean operands. If you pass in another type, it will be converted to boolean; see the section [Automatic conversion to boolean](#) in the data type documentation.

and

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

or

Resolves to `true` if either operand is true.

! (not)

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

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

Related information

[Booleans](#) on page 520

Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.

Arithmetic operators

Arithmetic expressions resolve to numeric values. Except for the unary negative `-`, arithmetic operators take two numeric operands. If an operand is a string, it's converted to numeric form. The operation fails if a string can't be converted.

+ (addition)

Resolves to the sum of the two operands.

- (subtraction and negation)

When used with two operands, resolves to the difference of the two operands, left minus right. When used with one operand, returns the value of subtracting that operand from zero.

/ (division)

Resolves to the quotient of the two operands, the left divided by the right.

*** (multiplication)**

Resolves to the product of the two operands. The asterisk is also used as a unary splat operator for arrays (see below).

% (modulo)

Resolves to the remainder of dividing the left operand by the right operand:

```
5 % 2    # resolves to 1
32 % 7   # resolves to 4
```

<< (left shift)

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

```
4 << 3   # resolves to 32: 4 times two cubed
```

>> (right shift)

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

```
16 << 3    # resolves to 2: 16 divided by two cubed
```

Related information

[Numbers](#) on page 530

Numbers in the Puppet language are normal integers and floating point numbers.

Array operators

Array operators take arrays as operands, and, with the exception of * (unary splat), they resolve to array values.

*** (splat)**

The unary splat operator * accepts a single array value. If you pass it a scalar value, it converts the value to a single-element array first. The splat operator "unfolds" an array, resolving to a comma-separated list values representing the array elements. It's useful in places where a comma-separated list of values is valid, including:

- The arguments of a function call.
- The cases of a case statement.
- The cases of a selector statement.

If you use it in other contexts, it resolves to the array that was passed in.

For example:

```
$a = ['vim', 'emacs']
myfunc($a)    # Calls myfunc with a single argument, the array containing
               'vim' and 'emacs'
:
               # myfunc(['vim','emacs'])
myfunc(*$a)   # Calls myfunc with two arguments, 'vim' and 'emacs':
               # myfunc('vim','emacs')
```

Another example:

```
$a = ['vim', 'emacs']
$x = 'vim'
notice case $x {
  $a      : { 'an array with both vim and emacs' }
  *$a     : { 'vim or emacs' }
  default : { 'no match' }
}
```

<< (append)

Resolves to an array containing the elements in the left operand, with the right operand as its final element.

The left operand must be an array, and the right operand can be any data type. Appending adds only a single element to an array. To add multiple elements from one array to another, use the concatenation operator +.

Examples:

```
[1, 2, 3] << 4 # resolves to [1, 2, 3, 4] [1, 2, 3] << [4, 5] # resolves to
[1, 2, 3, [4, 5]]
```

The append operator does not change its operands; it creates a new value.

+ (concatenation)

Resolves to an array containing the elements in the left operand followed by the elements in the right operand.

Both operands must be arrays. If the left operand isn't an array, Puppet interprets + as arithmetic addition. If the right operand is a scalar value, it is converted to a single-element array first.

Hash values are converted to arrays instead of wrapped, so you must wrap them yourself.

Examples:

```
[1, 2, 3] + 1      # resolves to [1, 2, 3, 1]
[1, 2, 3] + [1]    # resolves to [1, 2, 3, 1]
[1, 2, 3] + [[1]]  # resolves to [1, 2, 3, [1]]
```

The concatenation operator does not change its operands; it creates a new value.

- (removal)

Resolves to an array containing the elements in the left operand, with every occurrence of elements in the right operand removed.

Both operands must be arrays. If the left operand isn't an array, Puppet interprets - as arithmetic subtraction. If the right operand is a scalar value, it is converted to a single-element array first.

Hash values aren't automatically wrapped in arrays, so you must always wrap them yourself.

Examples:

```
[1, 2, 3, 4, 5, 1, 1] - 1      # resolves to [2, 3, 4, 5]
[1, 2, 3, 4, 5, 1, 1] - [1]    # resolves to [2, 3, 4, 5]
[1, 2, 3, [1, 2]] - [1, 2]     # resolves to [3, [1, 2]]
[1, 2, 3, [1, 2]] - [[1, 2]]  # resolves to [1, 2, 3]
```

The removal operator does not change its operands; it creates a new value.

Related information

[Arrays](#) on page 516

Arrays are ordered lists of values. Resource attributes which accept multiple values (including the relationship metaparameters) generally expect those values in an array. Many functions also take arrays, including the iteration functions.

Hash operators

Hash operators accept hashes as their left operand, and hashes or specific kinds of arrays as their right operand. The expressions resolve to hash values.

+ (merging)

Resolves to a hash containing the keys and values in the left operand and the keys and values in the right operand. If a key is present in both operands, the final hash uses the value from the right. It does not merge hashes recursively; it only merges top-level keys.

The right operand can be one of the following:

- A hash
- An array with an even number of elements. Each pair is converted in order to a key-value hash pair.

Examples:

```
{a => 10, b => 20} + {b => 30} # resolves to {a => 10, b => 30}
{a => 10, b => 20} + {c => 30} # resolves to {a => 10, b => 30, c => 30}
{a => 10, b => 20} + [c, 30]   # resolves to {a => 10, b => 30, c => 30}
{a => 10, b => 20} + 30        # gives an error
```

```
{a => 10, b => 20} + [30]      # gives an error
```

The merging operator does not change its operands; it creates a new value.

- (removal)

Resolves to a hash containing the keys and values in the left operand, minus any keys that are also present in the right operand.

The right operand can be one of the following:

- A hash. The keys present in this hash will be absent in the final hash, regardless of whether that key has the same values in both operands. The key, not the value, determines whether it's removed.
- An array of keys.
- A single key.

Examples:

```
{a => first, b => second, c => 17} - {c => 17, a => "something else"} #
resolves to {b => second}
{a => first, b => second, c => 17} - {a => a, d => d}                  #
resolves to {b => second, c => 17}
{a => first, b => second, c => 17} - [c, a]                          #
resolves to {b => second}
{a => first, b => second, c => 17} - c                               #
resolves to {a => first, b => second}
```

The removal operator does not change its operands; it creates a new value.

Related information

[Hashes](#) on page 527

Hashes map keys to values, maintaining the order of the entries according to insertion order.

Assignment operator

Puppet has one assignment operator, `=`.

= (assignment)

The assignment operator sets the variable on the left side to the value on the right side. The expression resolves to the value of the right hand side. Variables can be set only one time, after which, attempts to set the variable to a new value cause an error.

Related information

[Variables](#) on page 380

Variables store values so that those values can be accessed in code later.

Conditional statements and expressions

Conditional statements let your Puppet code behave differently in different situations. They are most helpful when combined with facts or with data retrieved from an external source. Puppet supports *if* and *unless* statements, *case* statements, and *selectors*.

Examples

An *if* statement evaluates the given condition and, if the condition resolves to `true`, executes the given code. This example includes an *elsif* condition, and gives a warning if you try to include the `ntp` class on a virtual machine or on machine running macOS:

```
if $facts['is_virtual'] {
  warning('Tried to include class ntp on virtual machine; this node might be
  misclassified.')
}
```

```

} elsif $facts['os']['family'] == 'Darwin' {
  warning('This NTP module does not yet work on our Mac laptops.')
} else {
  include ntp
}

```

An `unless` statement takes a Boolean condition and an arbitrary block of Puppet code, evaluates the condition, and if the condition is false, execute the code block. This statement sets `$maxclient` to 500 unless the system memory is above the specified parameter.

```

unless $facts['memory']['system']['totalbytes'] > 1073741824 {
  $maxclient = 500
}

```

A `case` statement evaluates a list of cases against a control expression, and executes the first code block where the case value matches the control expression. This example declares a role class on a node, but which role class it declares depends on what operating system the node runs:

```

case $facts['os']['name'] {
  'RedHat', 'CentOS': {
    include role::redhat
  }
  /^(Debian|Ubuntu)$/ : {
    include role::debian
  }
  default: {
    include role::generic
  }
}

```

A `selector` statement is similar to a `case` statement, but instead of executing code, it returns a value. This example returns the value `'wheel'` for the specified operating systems, but the value `'root'` for all other operating systems:

```

$rootgroup = $facts['os']['family'] ? {
  'RedHat'           => 'wheel',
  /(Debian|Ubuntu)/ => 'wheel',
  default            => 'root',
}

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

```

if statements

An *"if" statement* takes a Boolean condition and an arbitrary block of Puppet code, and executes the code block only if the condition is true. Optionally, an `if` statement can include `elsif` and `else` clauses.

Behavior

Puppet's `if` statements behave much like those in any other language. The `if` condition is evaluated first and, if it is true, the `if` code block is executed. If it is false, each `elsif` condition (if present) is tested in order, and if all conditions fail, the `else` code block (if present) is executed. If none of the conditions in the statement match and there is no `else` block, Puppet does nothing and moves on. If statements executes a maximum of one code block.

In addition to executing the code in a block, an `if` statement also produces a value, so the `if` statement can be used wherever a value is allowed. The value of an `if` expression is the value of the last expression in the executed block, or `undef` if no block was executed.

Syntax

An `if` statement consists of:

- The `if` keyword.
- A condition (any expression resolving to a Boolean value).
- A pair of curly braces containing any Puppet code.
- Optionally: any number of `elsif` clauses, which are processed in order.
- Optionally: the `else` keyword and a pair of curly braces containing Puppet code.

An `elsif` clause consists of:

- The `elsif` keyword.
- A condition.
- A pair of curly braces containing any Puppet code.

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

Conditions

The condition of an `if` statement can be any expression that resolves to a Boolean value. This includes:

- [Variables](#)
- [Expressions](#), including arbitrarily nested `and` and `or` expressions
- [Functions](#) that return values

Expressions that resolve to non-Boolean values are automatically converted to Booleans. For more information, see the [Booleans documentation](#).

Regex capture variables

If you use the regular expression match operator in a condition, any captures from parentheses in the pattern are available inside the associated code block as numbered variables (for example, `$1`, `$2`), and the entire match is available as `$0`. This example captures any digits from a hostname such as `www01` and `www02`, and stores them in the `$1` variable:

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

Regex capture variables are different from other variables in a couple of ways:

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

Related information

[Scope](#) on page 586

A scope is a specific area of code that is partially isolated from other areas of code.

[Booleans](#) on page 520

Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.

unless statements

"Unless" statements work like reversed `if` statements. They take a Boolean condition and an arbitrary block of Puppet code, evaluate the condition, and if it is false, execute the code block. They cannot include `elsif` clauses.

Behavior

The condition is evaluated first and, if it is false, the code block is executed. If the condition is true, Puppet does nothing and moves on.

In addition to executing the code in a block, an `unless` statement is also an expression that produces a value, and it can be used wherever a value is allowed. The value of an `unless` expression is the value of the last expression in the executed block. If no block was executed, the value is `undef`.

Syntax

The general form of an `unless` statement is:

- The `unless` keyword.
- A condition (any expression resolving to a Boolean value).
- A pair of curly braces containing any Puppet code.
- Optionally: the `else` keyword and a pair of curly braces containing Puppet code.

You cannot include an `elsif` clause in an `unless` statement. If you do, compilation fails with a syntax error.

```
unless $facts['memory']['system']['totalbytes'] > 1073741824 {
  $maxclient = 500
}
```

Conditions

The condition of an `unless` statement can be any expression that resolves to a Boolean value. This includes:

- [Variables](#).
- [Expressions](#), including arbitrarily nested `and` and `or` expressions.
- [Functions](#) that return values.

Expressions that resolve to non-Boolean values are automatically converted to Booleans. For more information, see the [Booleans documentation](#).

Regex capture variables

Although `unless` statements receive regex capture variables like `if` statements, you wouldn't usually use one, because the code in the statement is executed only if the condition doesn't match anything. It generally makes more sense to use an `if` statement.

case statements

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

Behavior

Puppet compares the control expression to each of the cases, in the order they are listed (except for the top-most level default case, which always goes last). It executes the block of code associated with the first matching case, and ignores the remainder of the statement. Case statements execute a maximum of one code block. If none of the cases match, Puppet does nothing and moves on.

In addition to executing the code in a block, a `case` statement is also an expression that produces a value, and can be used wherever a value is allowed. The value of a `case` expression is the value of the last expression in the executed block. If no block was executed, the value is `undef`.

The control expression of a case statement can be any expression that resolves to a value. This includes:

- [Variables](#).
- [Expressions](#).
- [Functions](#) that return values.

Syntax

The general form of a case statement is:

- The `case` keyword.
- A control expression, which is any expression resolving to a value.
- An opening curly brace.
- Any number of possible matches, which consist of:
 - A case or comma-separated list of cases.
 - A colon.
 - A pair of curly braces containing any arbitrary Puppet code.
 - A closing curly brace case.

```
case $facts['os']['name'] {
  'RedHat', 'CentOS': { include role::redhat } # Apply the redhat class
  /^(Debian|Ubuntu)$/ : { include role::debian } # Apply the debian class
  default: { include role::generic } # Apply the generic class
}
```

Case matching

A case can be any expression that resolves to a value, for example, literal values, variables and function calls. You can use a comma-separated list of cases to associate multiple cases with the same block of code. To use values from a variable as cases, use the `*` splat operator to convert an array of values into a comma-separated list of values.

Depending on the data type of a case's value, Puppet uses one of following behaviors to test whether the case matches:

- Most data types, for example, strings and Booleans, are compared to the control value with the `==` equality operator, which is case-insensitive when comparing strings.
- Regular expressions are compared to the control value with the `=~` matching operator, which is case-sensitive. Regex cases only match strings.
- Data types, such as `Integer`, are compared to the control value with the `is` matching operator. This tests whether the control value is an instance of that data type.
- Arrays are recursively compared to the control value. First, Puppet checks whether the control and array are the same length, then each corresponding element is compared using these same case matching rules.
- Hashes compare each key-value pair. To match, the control value and the case must have the same keys, and each corresponding value is compared using these same case matching rules.
- The special value `default` matches anything, and unless nested inside an array or hash, is always tested last regardless of its position in the list.

Regex capture variables

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

```
case $trusted['hostname'] {
  /www(\d+)/: { notice("Welcome to web server number ${1}"); include
    role::web }
  default: { include role::generic }
}
```

This example captures any digits from a hostname such as `www01` and `www02` and store them in the `$1` variable.

Regex capture variables are different from other variables in a couple of ways:

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

Best practices

Case statements must have a default case:

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

Related information

[Values, data types, and aliases](#) on page 504

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

[Regular expressions](#) on page 533

A regular expression (sometimes shortened to “regex” or “regexp”) is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

[Scope lookup rules](#) on page 589

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

Selector expressions

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

Behavior

The entire selector expression is treated as a single value. Puppet compares the control expression to each of the cases, in the order they are listed (except for the `default` case, which always goes last). When it finds a matching case, it treats that value as the value of the expression and ignore the remainder of the expression. If none of the cases match, Puppet fails compilation with an error, unless a `default` case is also provided.

The control expression of a selector can be any expression that resolves to a value. This includes:

- [Variables](#).
- [Expressions](#).
- [Functions](#) that return values

Selectors can be used wherever a value is expected. This includes:

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

Tip: For readability sake, use selectors only in variable assignments.

Syntax

Selectors resemble a cross between a case statement and the ternary operator found in other languages. The general form of a selector is:

- A control expression, which is any expression resolving to a value.
- The ? (question mark) keyword.
- An opening curly brace.
- Any number of possible matches, each of which consists of:
 - A case.
 - The => (hash rocket) keyword.
 - A value, which can be any expression resolving to a value.
 - A trailing comma.
- A closing curly brace.

In this example, the value of `$rootgroup` is determined using the value of `$facts['os']['family']`:

```
$rootgroup = $facts['os']['family'] ? {
  'Redhat'           => 'wheel',
  /(Debian|Ubuntu)/  => 'wheel',
  default            => 'root',
}

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

Case matching

In selector statements, you cannot use lists of cases. If the control expression is a string and you need more than one case associated with a single value, use a regular expression. Otherwise, use a case statement instead of a selector, because case statements do allow lists of cases. For more information, see [Case statements](#).

Regex capture variables

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

```
puppet
$system = $facts['os']['name'] ? {
  /(RedHat|Debian)/ => "our system is ${1}",
  default           => "our system is unknown",
}
```

Regex capture variables are different from other variables in a couple of ways:

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

Related information

[Values, data types, and aliases](#) on page 504

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

[Regular expressions](#) on page 533

A regular expression (sometimes shortened to “regex” or “regexp”) is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

[Scope](#) on page 586

A scope is a specific area of code that is partially isolated from other areas of code.

Function calls

Functions are plug-ins, written in Ruby, that you can call during catalog compilation. A call to any function is an expression that resolves to a value. Most functions accept one or more values as arguments, and return a resulting value.

The Ruby code in the function can do any number of things to produce the final value, including:

- Evaluate templates.
- Do mathematical calculations.
- Look up values from an external source.
- Cause side effects that modify the catalog.
- Evaluate a provided block of Puppet code, possibly using the function's arguments to modify that code or control how it runs.

Puppet includes several built-in functions. More functions are available in modules, such as `puppetlabs-stdlib`, on the [Forge](#). You can also write custom functions and put them in your own modules.

An entire function call—including the name, arguments, and lambda—constitutes an expression. It resolves to a single value, and can be used anywhere a value of that type is accepted. A function call might also have an effect, such as adding a class to the catalog. You can also use function calls on their own, which causes their effects to occur while their value is ignored.

All functions run during catalog compilation, which means they can access code and data only from the primary Puppet server. To make changes to an agent node, you must use a resource; to collect data from an agent node, you must use a custom fact.

Each function defines how many arguments it takes, what data types it expects those arguments to be, what values it returns, and any effects it has. For details about any functions built into Puppet, see the [function reference](#). For details about a function included in a module, see that module's documentation.

Statement functions

Statement functions are a group of built-in functions that are used only for their effects, rather than for any values. Puppet recognizes only its built-in statements; it doesn't allow adding new statement functions as plugins. The major difference between statement functions and other functions is that you can omit parentheses when calling a statement function with at least one argument, such as `include apache`.

Statement functions return a value like any other function, but they always return a value of `undef`. The built-in statement functions are:

Catalog statements

`include`: Includes the specified classes in a catalog.

`require`: Includes the specified classes in the catalog and adds them as a dependency of the current class or defined resource

`contain`: Includes the specified classes in the catalog and contains them in the current class.

`tag`: Adds the specified tag or tags to the containing class or defined resource.

Logging statements

`debug`: Logs message at the debug level.

`info`: Logs message at the info level.

`notice`: Logs message at the notice level.

`warning`: Logs message at the warning level

`err`: Logs message at the error level.

Failure statements

`fail`: Logs the error message and terminates compilation.

Related information

[Custom functions overview](#) on page 484

Puppet includes many built-in functions, and more are available in modules on the Forge. You can also write your own custom functions.

Functions syntax

Like any expression, a function call can be used anywhere the value it returns would be allowed. Function calls can also stand on their own, to cause their side effects, but ignore their returned value.

There are two ways to call functions in the Puppet language: *prefix calls* as in `template("ntp/ntp.conf.erb")`, and *chained calls* as in `"ntp/ntp.conf.erb".template`. There's also a modified form of prefix call that can only be used with certain functions.

The two function call styles have exactly the same capabilities, so you can choose whichever one is more readable. In general:

- For functions that take many arguments, prefix calls are easier to read.
- For functions that take one normal argument and a lambda, chained calls are easier to read.
- For a series of functions where each takes the last one's result as its argument, chained calls are easier to read, especially if at least one of those functions accepts a lambda.

Most functions have short, one-word names. The modern function API also allows qualified function names like `mymodule::myfunction`. Functions must always be called with their full names; you can't shorten a qualified function name.

Prefix function calls

You call a function in the prefix style by writing its name and providing a list of arguments in parentheses. The general form of a prefix function call is:

```
function_name(argument, argument, ...) |$parameter, $parameter, ...| { code
  block }
```

- The full name of the function, as an unquoted word.
- An opening parenthesis (. Parentheses are optional when you're calling a built-in statement function with at least one argument, as in `include apache`. They're mandatory in all other cases.
- Zero or more *arguments*, separated by commas. Arguments can be any expression that resolves to a value. See each function's docs for the number of its arguments and their data types. Use the splat array operator `*` to convert an array into a comma-separated list of arguments.
- A closing parenthesis) if an opening parenthesis was used.
- Optionally, a lambda (code block), if the function accepts one.

In the following example, `template`, `include`, and `each` are all functions. The `template` function is used for its return value, `include` adds a class to the catalog, and `each` runs a block of code several times with different values.

```
file {"/etc/ntp.conf":
  ensure => file,
  content => template("ntp/ntp.conf.erb"), # function call; resolves to a
  string
}
```

```
include apache # function call; modifies catalog

$binaries = [
  "factor",
  "hiera",
  "mco",
  "puppet",
  "puppetserver",
]

# function call with lambda; runs block of code several times
each($binaries) |$binary| {
  file {"/usr/bin/$binary":
    ensure => link,
    target => "/opt/puppetlabs/bin/$binary",
  }
}
```

Chained function calls

Alternatively, you call a function in the chained style by writing its first argument, a period, and the name of the function. The general form of a chained function call is:

```
argument.function_name(argument, ...) |$parameter, $parameter, ...| { code
  block }
```

- The first argument of the function, which can be any expression that resolves to a value.
- A period .
- The full name of the function, as an unquoted word.
- Optionally, parentheses containing a comma-separated list of additional arguments, starting with the second argument (because you already passed in the first argument). Use the splat array operator * to convert an array to a comma-separated list of arguments.
- Optionally, a lambda (code block), if the function accepts one.

In the following example, `template`, `include`, and `each` are all functions. The `template` function is used for its return value, `include` adds a class to the catalog, and `each` runs a block of code several times with different values.

```
puppet
file {"/etc/ntp.conf":
  ensure => file,
  content => "ntp/ntp.conf.erb".template, # function call; resolves to a
  string
}

apache.include # function call; modifies catalog

$binaries = [
  "factor",
  "hiera",
  "mco",
  "puppet",
  "puppetserver",
]

# function call with lambda; runs block of code several times
$binaries.each |$binary| {
  file {"/usr/bin/$binary":
    ensure => link,
    target => "/opt/puppetlabs/bin/$binary",
  }
}
```



```
}
```

Related information

[Lambdas](#) on page 572

Lambdas are blocks of Puppet code passed to functions. When a function receives a lambda, it provides values for the lambda's parameters and evaluates its code. If you use other programming languages, think of lambdas as anonymous functions that are passed to other functions.

[Expressions and operators](#) on page 422

Expressions are statements that resolve to values. You can use expressions almost anywhere a value is required. Expressions can be compounded with other expressions, and the entire combined expression resolves to a single value.

[Values, data types, and aliases](#) on page 504

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

[Array operators](#) on page 429

Array operators take arrays as operands, and, with the exception of `*` (unary splat), they resolve to array values.

Node definitions

A node definition, also known as a node statement, is a block of Puppet code that is included only in matching nodes' catalogs. This allows you to assign specific configurations to specific nodes.

Put node definitions in the main manifest, which can be a single `site.pp` file, or a directory containing many files.

If the main manifest contains at least one node definition, it must have one for *every* node. Compilation for a node fails if a node definition for it cannot be found. Either specify no node definitions, or use the `default` node definition, as described below, to avoid this situation.

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

Node definitions create an anonymous scope that can override variables and defaults from top scope.

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

Node definitions are an optional feature of Puppet. You can use them instead of or in combination with an [external node classifier \(ENC\)](#). Alternatively, you can use conditional statements with facts to classify nodes. Unlike more general conditional structures, node definitions match nodes only by name. By default, the name of a node is its certname, which defaults to the node's fully qualified domain name.

Although you can use node definitions in conjunction with an ENC, it's simpler to choose one method or the other. If you do use them together, Puppet merges their data as follows:

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

Syntax

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

- The node keyword.
- The node definition name: a quoted string, a regular expression, or `default`.
- An opening curly brace.
- Any mixture of class declarations, variables, resource declarations, collectors, conditional statements, chaining relationships, and functions.
- A closing curly brace.

In the following example, only `www1.example.com` receives the `apache` and `squid` classes, and only `db1.example.com` receives the `mysql` class:

```
# <ENVIRONMENTS DIRECTORY>/<ENVIRONMENT>/manifests/site.pp
node 'www1.example.com' {
  include common
  include apache
  include squid
}
node 'db1.example.com' {
  include common
  include mysql
}
```

A node definition name must be one of the following:

- A quoted string containing only letters, numbers, underscores (`_`), hyphens (`-`), and periods (`.`).
- A regular expression.
- The bare word `default`. If no other node definition matches a given node, the `default` node definition will be used for that node.

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

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

If you use a regular expression for a node definition name, it also has the potential to match multiple nodes. For example, the following node definition matches `www1`, `www13`, and any other node whose name consists of `www` and one or more digits:

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

The following example of a regex node definition name matches `one.example.com` and `two.example.com`, but no other nodes:

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

Important: Make sure all of your node definition name regexes match non-overlapping sets of node names. If a node's name matches more than one regex, Puppet makes no guarantee about which matching definition it will get.

You can use regex capture variables by enclosing parts of your regex node definition name in parentheses (`()`), and then referencing them in order as `$1`, `$2` and so on, as variables within the body of the node definition. For example:

```
node /^www(\d+)/ {
  $wwwnumber = $1 #assigns the value of the (\d+) from a regex match to the
  variable $wwwnumber
}
```

Matching

A given node gets the contents of only **one** node definition, even if multiple node definitions could match its name. Puppet does the following checks, in this order, until it finds one that matches:

1. If there is a node definition with the node's exact name, Puppet uses it.

2. If there is a regular expression node definition that matches the node's name, Puppet uses it. If more than one regex node matches, Puppet uses one of them, but we can't predict which. Make your node definition name regexes non-overlapping to avoid this problem.
3. By default, the primary server's `strict_hostname_checking` is set to `true`, which means the nodes are always matched by the `certname`.

If `strict_hostname_checking` is set to `false` and the node's name looks like a fully qualified domain name (it has multiple period-separated groups of letters, numbers, underscores, and dashes), Puppet chops off the final group and starts again at step 1. This is also true for `fqdn`. If the `fqdn` fact is not found, it will combine the `hostname` and `domain` facts.

4. Puppet uses the `default` node.

For example, when compiling a catalog for a node with `certname www01.example.com`, with fuzzy checking, Puppet looks for a node definition with the following name, in this order:

- `www01.example.com`
- A regex that matches `www01.example.com`
- `www01.example`
- A regex that matches `www01.example`
- `www01`
- A regex that matches `www01`
- `default`

If it doesn't find one, catalog compilation fails. It's a good idea to always have a `default` node definition.

Related information

[Variables](#) on page 380

Variables store values so that those values can be accessed in code later.

[Conditional statements and expressions](#) on page 431

Conditional statements let your Puppet code behave differently in different situations. They are most helpful when combined with facts or with data retrieved from an external source. Puppet supports *if* and *unless* statements, *case* statements, and *selectors*.

[Key configuration settings](#) on page 59

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

[Main manifest directory](#) on page 298

Puppet starts compiling a catalog either with a single manifest file or with a directory of manifests that are treated like a single file. This starting point is called the *main manifest* or *site manifest*.

[Scope](#) on page 586

A scope is a specific area of code that is partially isolated from other areas of code.

Facts and built-in variables

Before requesting a catalog for a managed node, or compiling one with `puppet apply`, Puppet collects system information, called *facts*, by using the *Facter* tool. The facts are assigned as values to variables that you can use anywhere in your manifests. Puppet also sets some additional special variables, called *built-in variables*, which behave a lot like facts.

Puppet code can access the following facts when compiling a catalog:

- [Core facts](#) from *Facter*.
- [Custom facts](#) and [external facts](#) that are present in your modules.

To see the fact values for a node, run `facter -p` on the command line, or browse facts on node detail pages in the Puppet Enterprise console. You can also use the PuppetDB [API](#) to explore or build tools to search and report on your infrastructure's facts.

Puppet honors fact values of any [data type](#). It does not convert Boolean, numeric, or structured facts to strings.

- [Accessing facts from Puppet code](#) on page 444

When you write Puppet code, you can access facts in two ways: with the `$fact_name` syntax, or with the `$facts['fact_name']` hash.

- [Built-in variables](#) on page 445

In addition to Factor's core facts and custom facts, Puppet creates several variables for a node to facilitate managing it. These variables are called trusted facts, server facts, agent facts, server variables, and compiler variables.

Accessing facts from Puppet code

When you write Puppet code, you can access facts in two ways: with the `$fact_name` syntax, or with the `$facts['fact_name']` hash.

Using the `$fact_name` syntax

Facts appear in Puppet as top-scope variables. They can be accessed in manifests as `$fact_name`.

For example:

```
if $osfamily == 'RedHat' {
  # ...
}
```

Tip: When you code with this fact syntax, it's not immediately obvious that you're using a fact — someone reading your code needs to know which facts exist to guess that you're accessing a top-scope variable. To make your code easier for others to read, use the `$::fact_name` syntax as a hint, to show that it's accessing a top-scope variable.

Using the `$facts['fact_name']` hash syntax

Alternatively, facts are structured in a `$facts` hash, and your manifest code can access them as `$facts['fact_name']`. The variable name `$facts` is reserved, so local scopes cannot re-use it. Structured facts show up as a nested structure inside the `$facts` namespace, and can be accessed using Puppet's normal hash access syntax.

For example:

```
if $facts['os']['family'] == 'RedHat' {
  # ...
}
```

Accessing facts using this syntax makes for more readable and maintainable code, by making facts visibly distinct from other variables. It eliminates confusion that is possible when you use a local variable whose name happens to match that of a common fact.

Because of ambiguity with function invocation, the dot-separated access syntax that is available in Factor commands is not available with the `$facts` hash access syntax. However, you can instead use the `get` function built into core Puppet. For more information, see [README](#).

Improving performance by blocking or caching built-in facts

If Factor is slowing down your code, you can configure Factor to block or cache built-in facts. When a system has a lot of something — for example, mount points or disks — Factor can take a long time to collect the facts from each one. When this is a problem, you can speed up Factor's collection by configuring these settings in the `factor.conf` file:

- `blocklist` for blocking built-in facts you're uninterested in.
- `ttl`s for caching built-in facts you don't need retrieved frequently.

Related information

[Scope](#) on page 586

A scope is a specific area of code that is partially isolated from other areas of code.

[Variables](#) on page 380

Variables store values so that those values can be accessed in code later.

[Hashes](#) on page 527

Hashes map keys to values, maintaining the order of the entries according to insertion order.

[Configuring Facter with `facter.conf`](#) on page 238

The `facter.conf` file is a configuration file that allows you to cache and block fact groups and facts, and manage how Facter interacts with your system. There are four sections: `facts`, `global`, `cli` and `fact-groups`. All sections are optional and can be listed in any order within the file.

Built-in variables

In addition to Facter's core facts and custom facts, Puppet creates several variables for a node to facilitate managing it. These variables are called trusted facts, server facts, agent facts, server variables, and compiler variables.

Trusted facts

Normal facts are self-reported by the node, and nothing guarantees their accuracy. Trusted facts are extracted from the node's certificate, which can prove that the certificate authority checked and approved them, making them useful for deciding whether a given node can receive the sensitive data in its catalog.

Trusted facts is a hash that contains trusted data from the node's certificate. You can access the data using the syntax `$trusted['fact_name']`. The variable name `$trusted` is reserved, so local scopes cannot reuse it.

Keys in the <code>\$trusted</code> hash	Possible values
<code>authenticated</code>	<p>An indication of whether the catalog request was authenticated, as well as how it was authenticated. The value will be one of these:</p> <ul style="list-style-type: none"> <code>remote</code> for authenticated remote requests, as with agent-server Puppet configurations. <code>local</code> for all local requests, as with standalone Puppet apply nodes. <code>false</code> for unauthenticated remote requests, possible if your <code>auth.conf</code> configuration allows unauthenticated catalog requests.

Keys in the \$trusted hash	Possible values
certname	<p>The node's subject certificate name, as listed in its certificate. When first requesting its certificate, the node requests a subject certificate name matching the value of its certname setting.</p> <ul style="list-style-type: none"> • If <code>authenticated</code> is <code>remote</code>, the value is the subject certificate name extracted from the node's certificate. • If <code>authenticated</code> is <code>local</code>, the value is read directly from the <code>certname</code> setting. • If <code>authenticated</code> is <code>false</code>, the value will be an empty string.
domain	<p>The node's domain, as derived from its validated certificate name. The value can be empty if the certificate name doesn't contain a fully qualified domain name.</p>
extensions	<p>A hash containing any custom extensions present in the node's certificate. The keys of the hash are the extension OIDs. OIDs in the <code>ppRegCertExt</code> range appear using their short names, and other OIDs appear as plain dotted numbers. If no extensions are present, or <code>authenticated</code> is <code>local</code> or <code>false</code>, this is an empty hash.</p>
hostname	<p>The node's hostname, as derived from its validated certificate name</p>

A typical \$trusted hash looks something like this:

```
{
  'authenticated' => 'remote',
  'certname'      => 'web01.example.com',
  'domain'        => 'example.com',
  'extensions'    => {
    'pp_uuid'                => 'ED803750-
E3C7-44F5-BB08-41A04433FE2E',
    'pp_image_name'          => 'storefront_production'
    '1.3.6.1.4.1.34380.1.2.1' => 'ssl-termination'
  },
  'hostname'       => 'web01'
}
```

Here is some example Puppet code using a certificate extension:

```
if $trusted['extensions']['pp_image_name'] == 'storefront_production' {
  include private::storefront::private_keys
}
```

Here's an example of a `hiera.yaml` file using certificate extensions in a hierarchy:

```
---
version: 5
hierarchy:
  - name: "Certname"
    path: "nodes/{trusted.certname}.yaml"
  - name: "Original VM image name"
    path: "images/{trusted.extensions.pp_image_name}.yaml"
  - name: "Machine role (custom certificate extension)"
    path: "role/{trusted.extensions.'1.3.6.1.4.1.34380.1.2.1'}.yaml"
  - name: "Common data"
    path: "common.yaml"
```

Server facts

The `$server_facts` variable provides a hash of server-side facts that cannot be overwritten by client side facts. This is important because it enables you to get trusted server facts that could otherwise be overwritten by client-side facts.

For example, the primary Puppet server sets the global `$::environment` variable to contain the name of the node's environment. However, if a node provides a fact with the name `environment`, that fact's value overrides the server-set `environment` fact. The same happens with other server-set global variables, like `$::servername` and `$::serverip`. As a result, modules can't reliably use these variables for whatever their intended purpose was.

A warning is issued any time a node parameter is overwritten.

Here is an example `$server_facts` hash:

```
{
  serverversion => "4.1.0",
  servername    => "v85ix8blah.delivery.example.com",
  serverip      => "192.0.2.10",
  environment   => "production",
}
```

Agent facts

Puppet agent and Puppet apply both add several extra pieces of info to their facts before requesting or compiling a catalog. Like other facts, these are available as either top-scope variables or elements in the `$facts` hash.

Agent facts	Values
<code>\$clientcert</code>	The node's certname setting . This is self-reported; for the verified certificate name, use <code>\$trusted['certname']</code> .
<code>\$clientversion</code>	The current version of Puppet agent.
<code>\$puppetversion</code>	The current version of Puppet on the node.
<code>\$clientnoop</code>	The value of the node's noop setting (true or false) at the time of the run.

Agent facts	Values
<code>\$agent_specified_environment</code>	The value of the node's environment setting . If the primary server's node classifier specified an environment for the node, <code>\$agent_specified_environment</code> and <code>\$environment</code> can have different values. If no value was set for the environment setting (in <code>puppet.conf</code> or with <code>--environment</code>), the value of <code>\$agent_specified_environment</code> is <code>undef</code> . That is, it doesn't default to <code>production</code> like the setting does.

Server variables

Several variables are set by the primary Puppet server. These are most useful when managing Puppet with Puppet, for example, managing the `puppet.conf` file with a template. Server variables are **not** available in the `$facts` hash.

Server variables	Values
<code>\$environment</code> (also available to <code>puppet apply</code>)	The agent node's environment. Note that nodes can accidentally or purposefully override this with a custom fact; the <code>\$server_facts['environment']</code> variable always contains the correct environment, and can't be overridden.
<code>\$servername</code>	The primary server's fully-qualified domain name (FQDN). Note that this information is gathered from the primary server by <code>Facter</code> , rather than read from the config files. Even if the primary server's <code>certname</code> is set to something other than its FQDN, this variable still contains the server's FQDN.
<code>\$serverip</code>	The primary server's IP address.
<code>\$serverversion</code>	The current version of Puppet on the primary server.
<code>\$settings::<SETTING_NAME></code> (also available to <code>puppet apply</code>)	The value of any of the primary server's configuration settings . This is implemented as a special namespace and these variables must be referred to by their qualified names. Other than <code>\$environment</code> and <code>\$clientnoop</code> , the agent node's settings are not available in manifests. If you wish to expose them to the primary server, you must create a custom fact.
<code>\$settings::all_local</code>	Contains all variables in the <code>\$settings</code> namespace as a hash of <code><SETTING_NAME> => <SETTING_VALUE></code> . This helps you reference settings that might be missing, because a direct reference to such a missing setting raises an error when <code>--strict_variables</code> is enabled.

Compiler variables

Compiler variables are set in every local scope by the compiler during compilation. They are mostly used when implementing complex defined types. Compiler variables are **not** available in the `$facts` hash.

These variables are always considered defined, such as `strict_variables` setting always considers these variables to be defined, but their value is `undef` whenever no other value is applicable.

Compiler variables	Values
<code>\$module_name</code>	The name of the module that contains the current class or defined type.
<code>\$caller_module_name</code>	The name of the module in which the <i>specific instance</i> of the surrounding defined type was declared. This is useful when creating versatile defined types that will be reused by several modules.

Related information

[CSR attributes and certificate extensions](#) on page 211

When Puppet agent nodes request their certificates, the certificate signing request (CSR) usually contains only their certname and the necessary cryptographic information. Agents can also embed additional data in their CSR, useful for policy-based autosigning and for adding new trusted facts.

[Configuring Hiera](#) on page 249

The Hiera configuration file is called `hiera.yaml`. It configures the hierarchy for a given layer of data.

[About environments](#) on page 289

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

[Puppet settings](#) on page 56

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

[Writing custom facts](#) on page 230

A typical fact in Facter is an collection of several elements, and is written either as a simple value (“flat” fact) or as structured data (“structured” fact). This page shows you how to write and format facts correctly.

[Scope](#) on page 586

A scope is a specific area of code that is partially isolated from other areas of code.

[Defined resource types](#) on page 418

Defined resource types, sometimes called defined types or defines, are blocks of Puppet code that can be evaluated multiple times with different parameters.

Reserved words and acceptable names

You can use only certain characters for naming variables, modules, classes, defined types, and other custom constructs. Additionally, some words in the Puppet language are reserved and cannot be used as bare word strings or names.

Reserved words

Reserved words cannot be used as:

- Bare word strings—to use these words as strings, you must enclose them in quotes.
- Names for custom functions.
- Names for classes.
- Names for custom resource types or defined resource types.

In addition, do not:

- Use the name of any existing resource type or function as the name of a function.
- Use the name of any existing resource type as the name of a defined type.
- Use the name of any existing data type (such as `integer`) as the name of a defined type.

Table 1:

Reserved word	Description
<code>and</code>	Expression operator
<code>application</code>	Language keyword
<code>attr</code>	Reserved for future use
<code>case</code>	Language keyword
<code>component</code>	Reserved
<code>consumes</code>	Language keyword
<code>default</code>	Language keyword
<code>define</code>	Language keyword
<code>elsif</code>	Language keyword
<code>environment</code>	Reserved for symbolic namespace use
<code>false</code>	Boolean value
<code>function</code>	Language keyword
<code>if</code>	Language keyword
<code>import</code>	Former language keyword
<code>in</code>	Expression operator
<code>inherits</code>	Language keyword
<code>node</code>	Language keyword
<code>or</code>	Expression operator
<code>private</code>	Reserved for future use
<code>produces</code>	Language keyword
<code>regexp</code>	Reserved
<code>site</code>	Language keyword
<code>true</code>	Boolean value
<code>type</code>	Language keyword
<code>undef</code>	Special value
<code>unit</code>	Reserved
<code>unless</code>	Language keyword

Reserved class names

Puppet automatically creates two names that must not be used as class names elsewhere:

- `main`: Puppet creates a `main` class, which contains any resources not contained by any other class.
- `settings`: Puppet creates a `settings` namespace, which contains variables with the settings available to the primary server.

Additionally, the names of data types can't be used as class names:

- `any`, `Any`

- array, Array
- binary, Binary
- boolean, Boolean
- catalogentry, catalogEntry, CatalogEntry
- class, Class
- collection, Collection
- callable, Callable
- data, Data
- default, Default
- deferred, Deferred
- enum, Enum
- float, Float
- hash, Hash
- integer, Integer
- notundef, NotUndef
- numeric, Numeric
- optional, Optional
- pattern, Pattern
- resource, Resource
- regexp, Regexp
- runtime, Runtime
- scalar, Scalar
- semver, SemVer
- semVerRange, SemVerRange
- sensitive, Sensitive
- string, String
- struct, Struct
- timespan, Timespan
- timestamp, TImESTAMP
- tuple, Tuple
- type, Type
- undef, Undef
- variant, Variant

Reserved variable names

The following variable names are reserved. Unless otherwise noted, you can't assign values to them or use them as parameters in classes or defined types.

Table 2:

Reserved variable name	Description
\$0, \$1, and every other variable name consisting only of digits	These are regex capture variables automatically set by regular expression used in conditional statements. Their values do not persist outside their associated code block or selector value. Assigning these variables causes an error.
Top-scope Puppet built-in variables and facts	Built-in variables and facts are reserved at top scope, but you can safely reuse them at node or local scope. See built-in variables and facts for a list of these variables and facts.

Reserved variable name	Description
<code>\$facts</code>	Reserved for facts and cannot be reassigned at local scopes.
<code>\$trusted</code>	Reserved for facts and cannot be reassigned at local scopes.
<code>\$server_facts</code>	If enabled, this variable is reserved for trusted server facts and cannot be reassigned at local scopes.
<code>title</code>	Reserved for the title of a class or defined type.
<code>name</code>	Reserved for the name of a class or defined type.

Related information

[Facts and built-in variables](#) on page 443

Before requesting a catalog for a managed node, or compiling one with `puppet apply`, Puppet collects system information, called *facts*, by using the *Facter* tool. The facts are assigned as values to variables that you can use anywhere in your manifests. Puppet also sets some additional special variables, called *built-in variables*, which behave a lot like facts.

[Scope](#) on page 586

A scope is a specific area of code that is partially isolated from other areas of code.

Acceptable characters in names

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



CAUTION: In some cases, names containing unsupported characters might still work. Such cases are bugs and could cease to work at any time. Removal of these bug cases is not limited to major releases.

Classes and defined resource type names

The names of classes and defined resource types can consist of one or more *namespace* segments. Each namespace segment:

- Must begin with a lowercase letter.
- Can include lowercase letters.
- Can include digits.
- Can include underscores.

When you follow these rules, each namespace segment matches the following regular expression:

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

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

Multiple namespace segments are joined together in a class or defined type name with the double colon namespace separator: `::`. Class names with multiple namespaces must match the following regular expression:

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

Some words and class names are reserved and cannot be used as class or defined type names. Additionally, the filename `init.pp` is reserved for the class named after any given module, so you cannot use the name `<MODULE NAME>::init` for a class or defined type.

Variable names

Variable names are case-sensitive and must begin with a dollar sign (\$). Most variable names must start with a lowercase letter or an underscore. The exception is regex capture variables, which are named with only numbers.

Variable names can include:

- Uppercase and lowercase letters
- Numbers
- Underscores (`_`). If the first character is an underscore, access that variable only from its own local scope.

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

Optionally, the name of the very first namespace can be empty, representing the top namespace. The main reason to namespace this way is to indicate to anyone reading your code that you're accessing a top-scope variable, such as `::is_virtual`.

You can also use a regular expression for variable names. Short variable names match the following regular expression:

```
\A\[a-z0-9_][a-zA-Z0-9_]*\Z
```

Qualified variable names match the following regular expression:

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

Module names

Module names obey the same rules as individual namespace segments, just as in a class or defined type name. That is, each namespace segment:

- Must begin with a lowercase letter.
- Can include lowercase letters.
- Can include digits.
- Can include underscores.

When you follow these rules, each namespace segment matches the following regular expression:

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

Reserved words and class names cannot be used as module names.

Parameter names

Parameter names begin with a dollar sign prefix (\$). The parameter name after the prefix:

- Must begin with a lowercase letter.
- Can include lowercase letters.
- Can include digits.
- Can include underscores.

When you follow these rules, a parameter name matches the following regular expression:

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

Tag names

Tag names must begin with:

- A lowercase letter, or
- An number, or
- An underscore.

Tag names can include:

- Lowercase letters
- Uppercase letters
- Digits
- Underscores
- Colons
- Periods
- Hyphens

When you follow these rules, a tag name matches the following regular expression:

```
\A[[:alnum:]]+[:_:-]*\Z
```

Resource names

Resource titles can contain any characters whatsoever and are case-sensitive. Resource names, or the *namevar* attribute, might be limited by the system being managed. For example, most operating systems have limits on the characters permitted in the name of a user account. You are generally responsible for knowing the name limits on the platforms you manage.

Node names

Node names can contain:

- Letters
- Digits
- Periods
- Underscores
- Dashes

That is, node names match the regular expression:

```
/\A[a-z0-9._-]+\Z/
```

Environment names

Environment names can contain:

- Lowercase letters
- Numbers
- Underscores

That is, environment names match the regular expression:

```
\A[a-z0-9_]+\Z
```

Custom resources

A *resource* is the basic unit that is managed by Puppet. Each resource has a set of attributes describing its state. Some attributes can be changed throughout the lifetime of the resource, whereas others are only reported back but cannot be changed, and some can only be set one time during initial creation.

A custom resource allows you to interact with something external. Three common use cases for this are:

- Parsing a file
- Running a command line tool
- Communicating with an API

To gather information about a resource and to enact changes on it, Puppet requires a *provider* to implement interactions. The provider can have parameters that influence its operation. To describe all these parts to the infrastructure and the consumers, the resource *type* defines all the metadata, including the list of the attributes. The provider contains the code to `get` and `set` the system state.

If you are starting from scratch, or want a simple method for writing types and providers, use the Resource API. Built on top of Puppet core, the Resource API makes the type and provider development easier, cheaper, safer, faster, and better. If you need to maintain existing code, need multiple providers, or need access to the catalog, use the old low-level types and providers method.

- [Develop types and providers with the Resource API](#) on page 455

The recommended method to create custom types and providers is to use the Resource API, which is built on top of Puppet core. It is easier, faster, and safer than the old types and providers method.

- [Resource API reference](#) on page 458

Use this information to understand how the Resource API works: how the resource is defined in the type, how resource management is implemented in the provider, and some of the known limitations of the Resource API.

- [Low-level method for developing types and providers](#) on page 473

You can define custom resource types for managing your infrastructure by defining types and providers. This original method has largely been replaced by the Resource API method, but you might already have a setup that uses this low-level method, or your situation might fall into one of the Resource API limitations.

Develop types and providers with the Resource API

The recommended method to create custom types and providers is to use the Resource API, which is built on top of Puppet core. It is easier, faster, and safer than the old types and providers method.

To get started developing types and providers with the Resource API:

1. Download [Puppet Development Kit](#) (PDK) appropriate to your operating system and architecture.
2. Create a [new module](#) with PDK, or work with an existing PDK-enabled module. To create a new module, run `pdk new module <MODULE_NAME>` from the command line, specifying the name of the module. Respond to the dialog questions.
3. To add the `puppet-resource_api` gem and enable modern rspec-style mocking, open the `.sync.yml` file in your editor, and add the following content:

```
# .sync.yml
---
Gemfile:
  optional:
    ':development':
      - gem: 'puppet-resource_api'
spec/spec_helper.rb:
  mock_with: ':rspec'
```

4. Apply these changes by running `pdk update`

5. To create the required files for a new type and provider in the module, run: `pdk new provider <provider_name>`

You will get the following response:

```
$ pdk new provider foo
pdk (INFO): Creating '../example/lib/puppet/type/foo.rb' from template.
pdk (INFO): Creating '../example/lib/puppet/provider/foo/foo.rb' from
template.
pdk (INFO): Creating '../example/spec/unit/puppet/provider/foo/
foo_spec.rb' from template.
$
```

The three generated files are the type (resource definition), the provider (resource implementation), and the unit tests. The default template contains an example that demonstrates the basic workings of the Resource API. This allows the unit tests to run immediately after creating the provider, which looks like this:

```
$ pdk test unit
[#] Preparing to run the unit tests.
[#] Running unit tests.
Evaluated 4 tests in 0.012065973 seconds: 0 failures, 0 pending.
[#] Cleaning up after running unit tests.
$
```

Writing the type and provider

Write a *type* to describe the resource and define its metadata, and a *provider* to gather information about the resource and implement changes.

Writing the type

The type contains the shape of your resources. The template provides the necessary name and ensure attributes. You can modify their description and the name's type to match your resource. Add more attributes as you need.

```
# lib/puppet/type/yum.rb
require 'puppet/resource_api'

Puppet::ResourceApi.register_type(
  name: 'yum',
  docs: <<-EOS,
    This type provides Puppet with the capabilities to manage ...
  EOS
  attributes: {
    ensure: {
      type: 'Enum[present, absent]',
      desc: 'Whether this apt key should be present or absent on the
target system.',
      default: 'present',
    },
    name: {
      type: 'String',
      desc: 'The name of the resource you want to manage.',
      behaviour: :namevar,
    },
  },
)
```

The following keys are available for defining attributes:

- `type`: the Puppet 4 data type allowed in this attribute. You can use all data types matching `Scalar` and `Data`.
- `desc`: a string describing this attribute. This is used in creating the automated API docs with [puppet-strings](#).

- `default`: a default value used by the runtime environment; when the caller does not specify a value for this attribute.
- `behaviour` / `behavior`: how the attribute behaves. Available values include:
 - `namevar`: marks an attribute as part of the primary key or identity of the resource. A given set of `namevar` values must distinctively identify an instance.
 - `init_only`: this attribute can only be set during the creation of the resource. Its value is reported going forward, but trying to change it later leads to an error. For example, the base image for a VM or the UID of a user.
 - `read_only`: values for this attribute are returned by `get()`, but `set()` is not able to change them. Values for this should never be specified in a manifest. For example, the checksum of a file, or the MAC address of a network interface.
 - `parameter`: values for this attributes are not returned by `get()`. You can use this attribute to influence how the provider behaves. For example, you can influence the [managehome](#) attribute when creating a user.

Writing the provider

The provider is the most important part of your new resource, as it reads and enforces state. When you generate a provider with the `pdk new provider` command, PDK generates a provider file like this generated `yum.rb` file

```
require 'puppet/resource_api/simple_provider'

# Implementation for the yum type using the Resource API.
class Puppet::Provider::provider_name::Yum <
  Puppet::ResourceApi::SimpleProvider
  def get(_context)
    [
      {
        name: 'foo',
        ensure: 'present',
      },
      {
        name: 'bar',
        ensure: 'present',
      },
    ]
  end

  def create(context, name, should)
    context.notice("Creating '#{name}' with #{should.inspect}")
  end

  def update(context, name, should)
    context.notice("Updating '#{name}' with #{should.inspect}")
  end

  def delete(context, name)
    context.notice("Deleting '#{name}'")
  end
end
```

The optional `initialize` method can be used to set up state that is available throughout the execution of the catalog. This is most often used for establishing a connection when talking to a service, such as when you are managing a database.

The `get(context)` method returns a list of hashes describing the resources that are on the target system. The basic example would return an empty list. For example, these resources could be returned from this:

```
[
  {
```

```

    name: 'a',
    ensure: 'present',
  },
  {
    name: 'b',
    ensure: 'present',
  },
]

```

The `create`, `update`, and `delete` methods are called by the `SimpleProvider` base class to change the system as requested by the catalog. The `name` argument is the name of the resource that is being processed. `should` contains the attribute hash — in the same format as `get` returns — with the values in the catalog.

When adding Ruby code to a module, follow these guidelines:

- For small sized blocks of code, put the code in the provider.
- For medium sized blocks of code, put the code into a separate file in `lib/puppet_x/$forgeuser/$modulename.rb`. `puppet_x` is optional, but it helps keep the file name unique and reduces the risk of a file overwriting code from an unknown dependency.
- For large sized blocks of code, put the code in a separate gem.

Unit testing

The generated unit tests in `spec/unit/puppet/provider/<PROVIDER_NAME>_spec.rb` are evaluated when you run `pdck test unit`.

Resource API reference

Use this information to understand how the Resource API works: how the resource is defined in the type, how resource management is implemented in the provider, and some of the known limitations of the Resource API.

Resource definition: the type

A type is a definition of a resource that Puppet can manage. The definition contains the resource's configurable properties and the parameters used to access it.

To make the resource known to the Puppet ecosystem, its definition, or *type* needs to be registered with Puppet. For example:

```

Puppet::ResourceApi.register_type(
  name: 'apt_key',
  desc: <<-EOS,
    This type provides Puppet with the capabilities to manage GPG keys
    needed
    by apt to perform package validation. Apt has it's own GPG keyring that
    can
    be manipulated through the `apt-key` command.

    apt_key { '6F6B15509CF8E59E6E469F327F438280EF8D349F':
      source => 'http://apt.puppet.com/pubkey.gpg'
    }

    **Autorequires**:
    If Puppet is given the location of a key file which looks like an
    absolute
    path this type will autorequire that file.
  EOS
  attributes: {
    ensure: {
      type: 'Enum[present, absent]',
      desc: 'Whether this apt key should be present or absent on the target
system.'
    },
  },
  id: {

```

```

    type:      'Variant[Pattern[/\A(0x)?[0-9a-fA-F]{8}\Z/], Pattern[/\A(0x)?[0-9a-fA-F]{16}\Z/], Pattern[/\A(0x)?[0-9a-fA-F]{40}\Z/]]',
    behaviour: :namevar,
    desc:      'The ID of the key you want to manage.',
  },
  source: {
    type: 'String',
    desc: 'Where to retrieve the key from, can be a HTTP(s) URL, or a local file. Files get automatically required.',
  },
  # ...
  created: {
    type:      'String',
    behaviour: :read_only,
    desc:      'Date the key was created, in ISO format.',
  },
},
autorequire: {
  file:      '$source', # evaluates to the value of the `source` attribute
  package: 'apt',
},
)

```

The `Puppet::ResourceApi.register_type(options)` function takes the following keyword arguments:

- **name:** the name of the resource type.
- **desc:** a doc string that describes the overall working of the resource type, provides examples, and explains prerequisites and known issues.
- **attributes:** a hash mapping attribute names to their details. Each attribute is described by a hash containing the Puppet 4 data type, a desc string, a default value, and the behavior of the attribute: `namevar`, `read_only`, `init_only`, or a parameter.
 - **type:** the Puppet 4 data type allowed in this attribute.
 - **desc:** a string describing this attribute. This is used in creating the automated API docs with [puppet-strings](#).
 - **default:** a default value that the runtime environment uses when you don't specify a value.
 - **behavior/behaviour:** how the attribute behaves. Currently available values:
 - **namevar:** marks an attribute as part of the "primary key" or "identity" of the resource. A given set of namevar values needs to distinctively identify an instance.
 - **init_only:** this attribute can only be set when creating the resource. Its value is reported going forward, but trying to change it later leads to an error. For example, the base image for a VM or the UID of a user.
 - **read_only:** values for this attribute are returned by `get()`, but `set()` is not able to change them. Values for this should never be specified in a manifest. For example, the checksum of a file, or the MAC address of a network interface.
 - **parameter:** these attributes influence how the provider behaves, and cannot be read from the target system. For example, the target file on inifile, or the credentials to access an API.
- **autorequire, autobefore, autosubscribe, and autonotify:** a hash mapping resource types to titles. The titles must either be constants, or, if the value starts with a dollar sign, a reference to the value of an attribute. If the specified resources exist in the catalog, Puppet creates the relationships that are requested here.
- **features:** a list of API feature names, specifying which optional parts of this spec the provider supports. Currently defined features: `canonicalize`, `simple_get_filter`, and `supports_noop`. See provider types for details.

Composite namevars (**title_patterns**)

Each resource being managed must be identified by a unique title. Usually this is straightforward and a single attribute can be used to act as an identifier. But sometimes you need a composite of two attributes to uniquely identify the resource you want to manage.

If multiple attributes are defined with the `namevar` behavior, the type specifies `title_patterns` that tell the Resource API how to get at the attributes from the title. If `title_patterns` is not specified, a default pattern is applied and matches against the first declared `namevar`.

Note: The order of the `title_patterns` is important. Declare the most specific pattern first and end with the most generic.

Each title pattern contains:

- `pattern`, which is a Ruby regular expression containing named captures. The names of the captures must be that of the `namevar` attributes.
- `desc`, a short description of what the pattern matches for.

For example:

```
Puppet::ResourceApi.register_type(
  name: 'software',
  docs: <<-DOC,
    This type provides Puppet with the capabilities to manage ...
  DOC
  title_patterns: [
    {
      pattern: %r{^(?<package>.*[^-])-(?<manager>.*)$},
      desc: 'Where the package and the manager are provided with a hyphen
seperator',
    },
    {
      pattern: %r{^(?<package>.*)$},
      desc: 'Where only the package is provided',
    },
  ],
  attributes: {
    ensure: {
      type: 'Enum[present, absent]',
      desc: 'Whether this resource should be present or absent on the
target system.',
      default: 'present',
    },
    package: {
      type: 'String',
      desc: 'The name of the package you want to manage.',
      behaviour: :namevar,
    },
    manager: {
      type: 'String',
      desc: 'The system used to install the package.',
      behaviour: :namevar,
    },
  },
)
```

These match the first title pattern:

```
software { php-yum:
  ensure=>'present'
}

software { php-gem:
  ensure=>'absent'
}
```

This matches the second title pattern:

```
software { php:
  manager='yum'
  ensure=>'present'
}
```

Resource implementation: the provider

To make changes, a resource requires an implementation, or *provider*. It is the code used to retrieve, update and delete the resources of a certain type.

The two fundamental operations to manage resources are reading and writing system state. These operations are implemented as `get` and `set`. The implementation itself is a Ruby class in the `Puppet::Provider` namespace, named after the type using CamelCase.

Note: Due to the way Puppet autoload works, this is in a file called `puppet/provider/<type_name>/<type_name>.rb`. The class also has the CamelCased type name twice.

At runtime, the current and intended system states a specific resource. These are represented as Ruby hashes of the resource's attributes and applicable operational parameters:

```
class Puppet::Provider::AptKey::AptKey
  def get(context)
    [
      {
        name: 'name',
        ensure: 'present',
        created: '2017-01-01',
        # ...
      },
      # ...
    ]
  end

  def set(context, changes)
    changes.each do |name, change|
      is = change.has_key? :is ? change[:is] : get_single(name)
      should = change[:should]
      # ...
    end
  end
end
```

The `get` method reports the current state of the managed resources. It returns an enumerable of all existing resources. Each resource is a hash with attribute names as keys, and their respective values as values. It is an error to return values not matching the type specified in the resource type. If a requested resource is not listed in the result, it is considered to not exist on the system. If the `get` method raises an exception, the provider is marked as unavailable during the current run, and all resources of this type fails in the current transaction. The exception message is reported.

The `set` method updates resources to a new state. The `changes` parameter gets passed a hash of change requests, keyed by the resource's name. Each value is another hash with the optional `:is` and `:should` keys. At least one of the two must be specified. The values are of the same shape as those returned by `get`. After the `set`, all resources are in the state defined by the `:should` values.

A missing `:should` entry indicates that a resource will be removed from the system. Even a type implementing the `ensure => [present, absent]` attribute pattern must react correctly on a missing `:should` entry. An `:is` key might contain the last available system state from a prior `get` call. If the `:is` value is `nil`, the resources were not found by `get`. If there is no `:is` key, the runtime did not have a cached state available.

The `set` method should always return `nil`. Signaling progress through the logging utilities described below. If the `set` method throws an exception, all resources that should change in this call and haven't already been marked

with a definite state, are marked as failed. The runtime only calls the `set` method if there are changes to be made, especially when resources are marked with `noop => true` (either locally or through a global flag). The runtime does not pass them to `set`. See `supports_noop` for changing this behavior if required.

Both methods take a context parameter which provides utilities from the runtime environment, and is described in more detail there.

Implementing simple providers

In many cases, the resource type follows the conventional patterns of Puppet, and does not gain from the complexities around batch-processing changes. For those cases, the `SimpleProvider` class supplies a proven foundation that reduces the amount of code necessary to get going.

`SimpleProvider` requires that your type follows these common conventions:

- `name` is the name of your `namevar` attribute.
- `ensure` attribute is present and has the `Enum[absent, present]` type.

To start using `SimpleProvider`, inherit from the class like this:

```
require 'puppet/resource_api/simple_provider'

# Implementation for the wordarray type using the Resource API.
class Puppet::Provider::AptKey::AptKey < Puppet::ResourceApi::SimpleProvider
  # ...
```

Next, instead of the `set` method, the provider needs to implement the `create`, `update` or `delete` methods:

- `create(context, name, should)`: Called to create a resource.
 - `context`: provides utilities from the runtime environment.
 - `name`: the name of the new resource.
 - `should`: a hash of the attributes for the new instance.
- `update(context, name, should)`: Called to update a resource.
 - `context`: provides utilities from the runtime environment.
 - `name`: the name of the resource to change.
 - `should`: a hash of the desired state of the attributes.
- `delete(context, name)`: Called to delete a resource.
 - `context`: provides utilities from the runtime environment.
 - `name`: the name of the resource that to be deleted.

The `SimpleProvider` does basic logging and error handling.

Provider features

There are some use cases where an implementation provides a better experience than the default runtime environment provides. To avoid burdening the simplest providers with that additional complexity, these cases are hidden behind feature flags. To enable the special handling, the resource definition has a `feature` key to list all features implemented by the provider.

canonicalize

Allows the provider to accept a wide range of formats for values without confusing the user.

```
Puppet::ResourceApi.register_type(
  name: 'apt_key',
  features: [ 'canonicalize' ],
)

class Puppet::Provider::AptKey::AptKey
  def canonicalize(context, resources)
```

```

resources.each do |r|
  r[:name] = if r[:name].start_with?('0x')
    r[:name][2..-1].upcase
  else
    r[:name].upcase
  end
end
end
end

```

The runtime environment needs to compare user input from the manifest (the desired state) with values returned from `get` (the actual state), to determine whether or not changes need to be affected. In simple cases, a provider only accepts values from the manifest in the same format as `get` returns. No extra work is required, as a value comparison is enough. This places a high burden on the user to provide values in an unnaturally constrained format. In the example, the `apt_key` name is a hexadecimal number that can be written with, and without, the `'0x'` prefix, and the casing of the digits is irrelevant. A value comparison on the strings causes false positives if the user inputs format that does not match. There is no hexadecimal type in the Puppet language. To address this, the provider can specify the `canonicalize` feature and implement the `canonicalize` method.

The `canonicalize` method transforms its `resources` argument into the standard format required by the rest of the provider. The `resources` argument to `canonicalize` is an enumerable of resource hashes matching the structure returned by `get`. It returns all passed values in the same structure with the required transformations applied. It is free to reuse or recreate the data structures passed in as arguments. The runtime environment must use `canonicalize` before comparing user input values with values returned from `get`. The runtime environment always passes canonicalized values into `set`. If the runtime environment requires the original values for later processing, it protects itself from modifications to the objects passed into `canonicalize`, for example by creating a deep copy of the objects.

The `context` parameter is the same passed to `get` and `set`, which provides utilities from the runtime environment, and is described in more detail there.

Note: When the provider implements canonicalization, it aims to always log the canonicalized values. As a result of `get` and `set` producing and consuming canonically formatted values, it is not expected to present extra cost.

A side effect of these rules is that the canonicalization of the `get` method's return value must not change the processed values. Runtime environments may have strict or development modes that check this property.

For example, in the Puppet Discovery runtime environment it is bound to the `strict` setting, and follows the established practices:

```

# puppet resource --strict=error apt_key ensure=present
> runtime exception

```

```

# puppet resource --strict=warning apt_key ensure=present
> warning logged but values changed

```

```

# puppet resource --strict=off apt_key ensure=present
> values changed

```

simple_get_filter

Allows for more efficient querying of the system state when only specific parts are required.

```

Puppet::ResourceApi.register_type(
  name: 'apt_key',
  features: [ 'simple_get_filter' ],
)

class Puppet::Provider::AptKey::AptKey
  def get(context, names = nil)
    [

```

```

    {
      name: 'name',
      # ...
    },
  ],
end

```

Some resources are very expensive to enumerate. The provider can implement `simple_get_filter` to signal extended capabilities of the `get` method to address this. The provider's `get` method is called with an array of resource names, or `nil`. The `get` method must at least return the resources mentioned in the `names` array, but may return more. If the `names` parameter is `nil`, all existing resources should be returned. The `names` parameter defaults to `nil` to allow simple runtimes to ignore this feature.

The runtime environment calls `get` with a minimal set of names, and keeps track of additional instances returned to avoid double querying. To gain the most benefits from batching implementations, the runtime minimizes the number of calls into `get`.

supports_noop

When a resource is marked with `noop => true`, either locally or through a global flag, the standard runtime produces the default change report with a `noop` flag set. In some cases, an implementation provides additional information, for example commands that would get executed, or require additional evaluation before determining the effective changes, such as `exec`'s `onlyif` attribute. The resource type specifies the `supports_noop` feature to have `set` called for all resources, even those flagged with `noop`. When the `noop` parameter is set to `true`, the provider must not change the system state, but only report what it would change. The `noop` parameter should default to `false` to allow simple runtimes to ignore this feature.

```

Puppet::ResourceApi.register_type(
  name: 'apt_key',
  features: [ 'supports_noop' ],
)

class Puppet::Provider::AptKey::AptKey
  def set(context, changes, noop: false)
    changes.each do |name, change|
      is = change.has_key? :is ? change[:is] : get_single(name)
      should = change[:should]
      # ...
      do_something unless noop
    end
  end
end

```

remote_resource

Declaring this feature restricts the resource from being run locally. It is expected to execute all external interactions through the `context.transport` instance. The way that an instance is set up is runtime specific. Use `puppet/resource_api/transport/wrapper` as the base class for all devices:

```

# lib/puppet/type/nx9k_vlan.rb
Puppet::ResourceApi.register_type(
  name: 'nx9k_vlan',
  features: [ 'remote_resource' ],
  # ...
)

# lib/puppet/util/network_device/nexus/device.rb
require 'puppet'
require 'puppet/resource_api/transport/wrapper'
# force registering the transport schema

```



```
require 'puppet/transport/schema/device_type'

module Puppet::Util::NetworkDevice::Nexus
  class Device < Puppet::ResourceApi::Transport::Wrapper
    def initialize(url_or_config, _options = {})
      super('nexus', url_or_config)
    end
  end
end

# lib/puppet/provider/nx9k_vlan/nx9k_vlan.rb
class Puppet::Provider::Nx9k_vlan::Nx9k_vlan
  def set(context, changes, noop: false)
    changes.each do |name, change|
      is = change.has_key? :is ? change[:is] : get_single(name)
      should = change[:should]
      # ...
      context.transport.do_something unless noop
    end
  end
end
```

Runtime environment

The primary runtime environment for the provider is the Puppet agent, a long-running daemon process. The provider can also be used in the `puppet apply` command, a self contained version of the agent, or the `puppet resource` command, a short-lived command line interface (CLI) process for listing or managing a single resource type. Other callers that want to access the provider must imitate these environments.

The primary life cycle of resource management in each of these tools is the transaction, a single set of changes, for example a catalog or a CLI invocation. The provider's class is instantiated one time for each transaction. Within that class the provider defines any number of helper methods to support itself. To allow for a transaction to set up the prerequisites for a provider and be used immediately, the provider is instantiated as late as possible. A transaction usually calls `get` one time, and may call `set` any number of times to make changes.

The object instance that hosts the `get` and `set` methods can be used to cache ephemeral state during execution. The provider should not try to cache state outside of its instances. In many cases, such caching won't help as the hosting process only manages a single transaction. In long-running runtime environments like the agent, the benefit of the caching needs to be balanced with the cost of the cache at rest, and the lifetime of cache entries, which are only useful when they are longer than the regular `runinterval`.

The runtime environment has the following utilities to provide a uniform experience for its users.

Logging and reporting utilities

The provider needs to signal changes, successes, and failures to the runtime environment. The `context` is the primary way to do this. It provides a structured logging interface for all provider actions. Using this information, the runtime environments can do automatic processing, emit human readable progress information, and provide status messages for operators.

To provide feedback about the overall operation of the provider, the `context` has the usual set of [loglevel](#) methods that take a string, and pass that up to the runtime environments logging infrastructure. For example:

```
context.warning("Unexpected state detected, continuing in degraded mode.")
```

Results in the following message:

```
Warning: apt_key: Unexpected state detected, continuing in degraded mode.
```

Other common messages include:

- `debug`: Detailed messages to understand everything that is happening at runtime, shown on request.

- `info`: Regular progress and status messages, especially useful before long-running operations, or before operations that can fail, to provide context for interactive users.
- `notice`: Indicates state changes and other events of notice from the regular operations of the provider.
- `warning`: Signals error conditions that do not (yet) prohibit execution of the main part of the provider; for example, deprecation warnings, temporary errors.
- `err`: Signals error conditions that have caused normal operations to fail.
- `critical`, `alert`, `emerg`: Should not be used by resource providers.

In simple cases, a provider passes off work to an external tool, logs the details there, and then reports back to Puppet acknowledging these changes. This is called resource status signaling, and looks like this:

```
@apt_key_cmd.run(context, action, key_id)
context.processed(key_id, is, should)
```

It reports all changes from `is` to `should`, using default messages.

Providers that want to have more control over the logging throughout the processing can use the more specific `created(title)`, `updated(title)`, `deleted(title)`, `unchanged(title)` methods. To report the change of an attribute, the context provides a `attribute_changed(title, attribute, old_value, new_value, message)` method.

Most of those messages are expected to be relative to a specific resource instance, and a specific operation on that instance. To enable detailed logging without repeating key arguments, and to provide consistent error logging, the context provides logging context methods to capture the current action and resource instance:

```
context.updating(title) do
  if apt_key_not_found(title)
    context.warning('Original key not found')
  end

  # Update the key by calling CLI tool
  apt_key(...)

  context.attribute_changed('content', nil, content_hash,
    message: "Replaced with content hash #{content_hash}")
end
```

This results in the following messages:

```
Debug: Apt_key[F1D2D2F9]: Started updating
Warning: Apt_key[F1D2D2F9]: Updating: Original key not found
Debug: Apt_key[F1D2D2F9]: Executing 'apt-key ...'
Debug: Apt_key[F1D2D2F9]: Successfully executed 'apt-key ...'
Notice: Apt_key[F1D2D2F9]: Updating content: Replaced with content hash
E242ED3B
Notice: Apt_key[F1D2D2F9]: Successfully updated
```

In the case of an exception escaping the block, the error is logged appropriately:

```
Debug: Apt_key[F1D2D2F9]: Started updating
Warning: Apt_key[F1D2D2F9]: Updating: Original key not found
Error: Apt_key[F1D2D2F9]: Updating failed: Something went wrong
```

Logging contexts process all exceptions. A `StandardError` is assumed to be regular failures in handling resources, and are consumed after logging. Everything else is assumed to be a fatal application-level issue, and is passed up the stack, ending execution. See the [Ruby documentation](#) for details on which exceptions are not a `StandardError`.

The equivalent long-hand form of manual error handling:

```
context.updating(title)
begin
  unless title_got_passed_to_set(title)
    raise Puppet::DevError, 'Managing resource outside of requested set:
    %{title}'
  end

  if apt_key_not_found(title)
    context.warning('Original key not found')
  end

  # Update the key by calling CLI tool
  result = @apt_key_cmd.run(...)

  if result.exitstatus != 0
    context.error(title, "Failed executing apt-key #{...}")
  else
    context.attribute_changed(title, 'content', nil, content_hash,
    message: "Replaced with content hash #{content_hash}")
  end
  context.changed(title)
rescue Exception => e
  context.error(title, e, message: 'Updating failed')
  raise unless e.is_a? StandardError
end
```

This example is only for demonstration purposes. In the normal course of operations, providers should always use the utility functions.

The following methods are available:

- Block functions: these functions provide logging and timing around a provider's core actions. If the the passed `&block` returns, the action is recorded as successful. To signal a failure, the block should raise an exception explaining the problem:
 - `creating(titles, message: 'Creating', &block)`
 - `updating(titles, message: 'Updating', &block)`
 - `deleting(titles, message: 'Deleting', &block)`
 - `processing(title, is, should, message: 'Processing', &block)`: generic processing of a resource, produces default change messages for the difference between `is:` and `should:`.
 - `failing(titles, message: 'Failing', &block)`: unlikely to be used often, but provided for completeness. It always records a failure.
- Action functions:
 - `created(titles, message: 'Created')`
 - `updated(titles, message: 'Updated')`
 - `deleted(titles, message: 'Deleted')`
 - `processed(title, is, should)`: the resource has been processed. It produces default logging for the resource and each attribute
 - `failed(titles, message:)`: the resource has not been updated successfully
- Attribute Change notifications:
 - `attribute_changed(title, attribute, is, should, message: nil)`: notify the runtime environment that a specific attribute for a specific resource has changed. `is` and `should` are the original and the new value of the attribute. Either can be `nil`.

- Plain messages:
 - `debug(message)`
 - `debug(titles, message:)`
 - `info(message)`
 - `info(titles, message:)`
 - `notice(message)`
 - `notice(titles, message:)`
 - `warning(message)`
 - `warning(titles, message:)`
 - `err(message)`
 - `err(titles, message:)`

`titles` can be a single identifier for a resource or an array of values, if the following block batch processes multiple resources in one pass. If that processing is not atomic, providers should instead use the non-block forms of logging, and provide accurate status reporting on the individual parts of update operations.

A single `set()` execution may only log messages for instances that have been passed, as part of the changes to process. Logging for instances not requested to be changed causes an exception - the runtime environment is not prepared for other resources to change.

The provider is free to call different logging methods for different resources in any order it needs to. The only ordering restriction is for all calls specifying the same `title`. For example, the `attribute_changed` needs logged before that resource's action logging, and the `context` needs to be opened before any other logging for this resource.

Type definition

The provider can gain insight into the type definition through these `context.type` utility methods:

- `attributes`: returns a hash containing the type attributes and it's properties.
- `ensurable?`: returns `true` if the type contains the `ensure` attribute.
- `feature?(feature)`: returns `true` if the type supports a given provider feature.

For example:

```
# example from simple_provider.rb

def set(context, changes)
  changes.each do |name, change|
    is = if context.type.feature?('simple_get_filter')
          change.key?(:is) ? change[:is] : (get(context, [name]) || []).find
        { |r| r[:name] == name }
        else
          change.key?(:is) ? change[:is] : (get(context) || []).find { |r|
            r[:name] == name }
          end
    ...
  end
end
```

Resource API transports

A *transport* connects providers to remote resources, such as a device, cloud infrastructure, or a REST API.

The transport class contains the code for managing connections and processing information to and from the remote resource. The transport schema, similar to a resource type, describes the structure of the data that is passed for it to make a connection.

Transport implementation methods

When you are writing a transport class to manage remote resources, use the following methods as appropriate:

- `initialize(context, connection_info)`
 - The `connection_info` contains a hash that matches the schema. After you run the `initialize` method, the provider assumes that you have defined your transport in such a way as to ensure that it's ready for processing requests. The transport will report connection errors by throwing an exception, for example, if the network is unreachable or the credentials are rejected. In some cases, for example when the target is a REST API, no processing will happen during initialization.
- `verify(context)`
 - Use this method to check whether the transport can connect to the remote target. If the connection fails, the transport will raise an exception.
- `facts(context)`
 - Use this method to access the target and the facts hash which contains a subset of default facts from `Facter`, and more specific facts appropriate for the target.
- `close(context)`
 - Use this method to close the connection. Calling this method releases the transport so that you can't use it any more and frees up cache and operating system resources, such as open connections. For implementation quality, the library will ignore exceptions that are thrown.

Note: The `context` is the primary way to signal changes, successes, and failures to the runtime environment. For more information, see [Runtime environment](#).

An example of a transport class:

```
# lib/puppet/transport/device_type.rb
module Puppet::Transport
  # The main connection class to a Device endpoint
  class DeviceType
    def initialize(context, connection_info)
      # Add additional validation for connection_info
      # and pre-load a connection if it is useful for your target
    end

    def verify(context)
      # Test that transport can talk to the remote target
    end

    def facts(context)
      # Access target, return a Facter facts hash
    end

    def close(context)
      # Close connection, free up resources
    end
  end
end
```

An example of a corresponding schema:

```
# lib/puppet/transport/device_type.rb
Puppet::ResourceAPI.register_transport(
  name: 'device_type', # points at class Puppet::Transport::DeviceType
  desc: 'Connects to a device_type',
  connection_info: {
    host: {
      type: 'String',
      desc: 'The host to connect to.',
    }
  }
)
```

```

    },
    user: {
      type: 'String',
      desc: 'The user.',
    },
    password: {
      type: 'String',
      sensitive: true,
      desc: 'The password to connect.',
    },
    enable_password: {
      type: 'String',
      sensitive: true,
      desc: 'The password escalate to enable access.',
    },
    port: {
      type: 'Integer',
      desc: 'The port to connect to.',
    },
  },
)

```

If the following attributes apply to your target, use these names for consistency across transports:

- `uri`: use to specify which URL to connect to.
- `host`: use to specify an IP or address to connect to.
- `protocol`: use to specify which protocol the transport uses, for example `http`, `https`, `ssh` or `tcp`.
- `user`: the user you want the transport to connect as.
- `port`: the port you want the transport to connect to.

Do not use the following keywords in when writing the `connection_info`:

- `name`
- `path`
- `query`
- `run-on`
- `remote-transport`
- `remote-*`
- `implementations`

To ensure that the data the schema passes to the implementation is handled securely, set password attributes to `sensitive: true`. Attributes marked with the `sensitive` flag allow a user interface based on this schema to make appropriate presentation choices, such as obscuring the password field. Values that you've marked sensitive are passed to the transport wrapped in the type `Puppet::Pops::Types::PSensitiveType::Sensitive`. This keeps the value from being logged or saved inadvertently while it is being transmitted between components. To access the sensitive value within the transport, use the `unwrap` method, for example, `connection_info[:password].unwrap`.

Errors and retry handling in transport implementation

The Resource API does not put many constraints on when and how a transport can fail. The remote resource you are connecting to will have it's own device specific connection and error handling capabilities. Be aware of the following issues that may arise:

- Your initial connection might fail. To retry making the connection, verify whether you have network problems or whether there has been a service restart of the target that you are trying to connect to. As part of the retry logic, the transport avoids passing these issues to other parts of your system and waits up to 30 seconds for a single target to recover. When you execute a retry, the transport logs transient problems at the `notice` level.
- After you make your connection and have run the `initialize` method, the transport might apply deeper validation to the passed connection information — like mutual exclusivity of optional values, for example, `password` or `key`

— and throw an `ArgumentError`. The transport then tries to establish a connection to the remote target. If this fails due to unrecoverable errors, it throws another exception.

- The `verify` and `facts` methods, like `initialize`, throw exceptions only when unrecoverable errors are encountered, or when the retry logic times out.

Port your existing device code to transports

If you have old code that uses `Device`, port it by updating your code with the following replacements as appropriate:

- Move the device class to `Puppet::Transport`
- Change `Util::NetworkDevice::NAME::Device` to `ResourceApi::Transport::NAME`
- Change the initialization to accept and process a `connection_info` hash.
- When accessing the `connection_info` in your new transport, change all string keys to symbols, for example, `name` to `:name`.
- Add context as the first argument to your `initialize` and `facts` method.
- Change `puppet/util/network_device/NAME/device` to `puppet/transport/NAME`
- Replace calls to [Puppet logging](#) with calls to the context logger

Note: If you can't port your code at this time, your providers can still access your `Device` through `context.device`, but you won't be able to make use of the functionality in [Bolt plans](#).

After porting your code, you will have a transport class and a shim `Device` class that connects your transport in a way that Puppet understands. Specify the transport name in the `super` call to make the connection:

```
# lib/puppet/type/nx9k_vlan.rb
Puppet::ResourceApi.register_type(
  name: 'nx9k_vlan',
  features: [ 'remote_resource' ],
  # ...
)

# lib/puppet/util/network_device/nexus/device.rb
require 'puppet/resource_api/transport/wrapper'
# force registering the transport
require 'puppet/transport/schema/nexus'

module Puppet::Util::NetworkDevice::Nexus
  class Device < Puppet::ResourceApi::Transport::Wrapper
    def initialize(url_or_config, _options = {})
      super('nexus', url_or_config)
    end
  end
end
```

Note: Agent versions 6.0 to 6.3 are incompatible with this way of executing remote content. These versions will not be supported after support [for PE 2019.0 ends](#).

Resource API limitations

This Resource API is not a full replacement for the original low-level types and providers method. Here is a list of the current limitations. If they apply to your situation, the low-level types and providers method might be a better solution. The goal of the new Resource API is not to be a replacement of the prior one, but to be a simplified way to get the same results for the majority of use cases.

You can't have multiple providers for the same type

The low-level type and provider method allows multiple providers for the same resource type. This allows the creation of abstract resource types, such as packages, which can span multiple operating systems. Automatic selection of an OS-appropriate provider means less work for the user, as they don't have to address whether the package needs to be managed using `apt` or `yum` in their code .

Allowing multiple providers means more complexity and more work for the type or provider developer, including:

- attribute sprawl
- disparate feature sets between the different providers for the same abstract type
- complexity in implementation of both the type and provider pieces stemming from the two issues above

The Resource API does not implement support for multiple providers.

If you want support for multiple providers for a given type, your options are:

- Use the older, more complex type and provider method, or
- Implement multiple similar types using the Resource API, and select the platform-appropriate type in Puppet code. For example:

```
define package (
  Ensure $ensure,
  Enum[apt, rpm] $provider, # have a hiera 5 dynamic binding to a function
                             # choosing a sensible default for the current system
  Optional[String] $source = undef,
  Optional[String] $version = undef,
  Optional[Hash] $options = { },
) {
  case $provider {
    apt: {
      package_apt { $title:
        ensure      => $ensure,
        source       => $source,
        version      => $version,
        *            => $options,
      }
    }
    rpm: {
      package_rpm { $title:
        ensure => $ensure,
        source => $source,
        *      => $options,
      }
      if defined($version) { fail("RPM doesn't support \"$version\"") }
      # ...
    }
  }
}
```

Only built-in Puppet 4 data types are available

Currently, only built-in Puppet 4 data types are usable. This is because the type information is required on the agent, but Puppet has not made it available yet. Even after that is implemented, modules have to wait until the functionality is widely available before being able to rely on it.

There is no catalog access

There is no way to access the catalog from the provider. Several existing types rely on this to implement advanced functionality. Some of these use cases would be better off being implemented as "external" catalog transformations, instead of munging the catalog from within the compilation process.

No logging for unmanaged instances

Previously, the provider could provide log messages for resource instances that were not passed into the `set` call. In the current implementation, these causes an error.

Automatic relationships constrained to consts and attribute values

The Puppet 3 type API allows arbitrary code execution for calculating automatic relationship targets. The Resource API is more restrained, but allows understanding the type's needs by inspecting the metadata.

Low-level method for developing types and providers

You can define custom resource types for managing your infrastructure by defining types and providers. This original method has largely been replaced by the Resource API method, but you might already have a setup that uses this low-level method, or your situation might fall into one of the Resource API limitations.

Using this types and providers method, you can add new resource types to Puppet. This section describes what types and providers are, how they interact, and how to develop them.

Low-level Puppet types and providers development is done in Ruby. Previous experience with Ruby is helpful. Alternatively, opt for the more straightforward Resource API method for developing types and providers.

The internals of how types are created have changed over Puppet's lifetime, and this documentation describes effective development methods, skipping over all the things you can but probably shouldn't do.

When making a new resource type, you create two things:

- The type definition, which is a model of the resource type. It defines what parameters are available, handles input validation, and determines what features a provider can (or should) provide.
- One or more providers for that type. The provider implements the type by translating its capabilities into specific operations on a system. For example, the `package` type has `yum` and `apt` providers which implement package resources on Red Hat-like and Debian-like systems, respectively.

Deploying types and providers

To use new types and providers:

1. The type and providers must be present in a module on the primary server. Like other types of plug-in (such as custom functions and custom facts), they go in the module's `lib` directory:
 - Type files: `lib/puppet/type/<TYPE NAME>.rb`
 - Provider files: `lib/puppet/provider/<TYPE NAME>/<PROVIDER NAME>.rb`
 2. If you are using an agent-server deployment, each agent node must have its `pluginsync` setting in `puppet.conf` set to `true`, which is the default. In serverless Puppet, using `puppet apply`, the `pluginsync` setting is not required, but the module that contains the type and providers must be present on each node.
- [Type development](#) on page 474

When you define a resource type, focus on what the resource can do, not how it does it.

- [Provider development](#) on page 480

Providers are back-ends that support specific implementations of a given resource type, particularly for different platforms. Not all resource types have or need providers, but any resource type concerned about portability will likely need them.

Related information

[Plug-ins in modules](#) on page 596

Puppet supports several kinds of plug-ins, which are distributed in modules. These plug-ins enable features such as custom facts and functions for managing your nodes. Modules that you download from the Forge can include these kinds of plug-ins, and you can also develop your own.

[Custom resources](#) on page 454

A *resource* is the basic unit that is managed by Puppet. Each resource has a set of attributes describing its state. Some attributes can be changed throughout the lifetime of the resource, whereas others are only reported back but cannot be changed, and some can only be set one time during initial creation.

[Resource API limitations](#) on page 471

This Resource API is not a full replacement for the original low-level types and providers method. Here is a list of the current limitations. If they apply to your situation, the low-level types and providers method might be a better

solution. The goal of the new Resource API is not to be a replacement of the prior one, but to be a simplified way to get the same results for the majority of use cases.

Type development

When you define a resource type, focus on what the resource can do, not how it does it.

Note: Unless you are maintaining existing type and provider code, or the Resource API limitations affect you, use the Resource API to create custom resource types, instead of this method.

Creating types

Types are created by calling the `newtype` method on the `Puppet::Type` class:

```
# lib/puppet/type/database.rb
Puppet::Type.newtype(:database) do
  @doc = "Create a new database."
  # ... the code ...
end
```

The name of the type is the only required argument to `newtype`. The name must be a [Ruby symbol](#), and the name of the file containing the type must match the type's name.

The `newtype` method also requires a block of code, specified with either curly braces (`{ ... }`) or the `do ... end` syntax. The code block implements the type, and contains all of the properties and parameters. The block will not be passed any arguments.

You can optionally specify a self-refresh option for the type by putting `:self_refresh => true` after the name. Doing so causes resources of this type to refresh (as if they had received an event through a notify-subscribe relationship) whenever a change is made to the resource. A notable use of this option is in the core `mount` type.

Documenting types

Write a description for the custom resource type in the type's `@doc` instance variable. The description can be extracted by the `puppet doc --reference type` command, which generates a complete type reference which includes your new type, and by the `puppet describe` command, which outputs information about specific types.

Write the description as a string in standard Markdown format. When the Puppet tools extract the string, they strip the greatest common amount of leading whitespace from the front of each line, excluding the first line. For example:

```
Puppet::Type.newtype(:database) do
  @doc = %q{Creates a new database. Depending
    on the provider, this might create relational
    databases or NoSQL document stores.

    Example:

        database { 'mydatabase':
          ensure => present,
          owner  => root,
        }
      }
end
```

In this example, any whitespace would be trimmed from the first line (in this case, it's zero spaces), then the greatest common amount would be trimmed from remaining lines. Three lines have four leading spaces, two lines have six, and two lines have eight, so four leading spaces would be trimmed from each line. This leaves the example code block indented by four spaces, and thus doesn't break the Markdown formatting.

Properties and parameters

The bulk of a type definition consists of properties and parameters, which become the resource attributes available when declaring a resource of the new type.

The difference between a property and a parameter is subtle but important:

- Properties correspond to something measurable on the target system. For example, the UID and GID of a user account are properties, because their current state can be queried or changed. In practical terms, setting a value for a property causes a method to be called on the provider.
- Parameters change how Puppet manages a resource, but do not necessarily map directly to something measurable. For example, the `user` type's `managehome` attribute is a parameter — its value affects what Puppet does, but the question of whether Puppet is managing a home directory isn't an innate property of the user account.

Additionally, there are a few special attributes called [metaparameters](#), which are supported by all resource types. These don't need to be handled when creating new types; they're implemented elsewhere.

A type definition typically has multiple properties, and must have at least one parameter.

Properties

A custom type's properties are at the heart of defining how the resource works. In most cases, it's the properties that interact with your resource's providers.

If you define a property named `owner`, then when you are retrieving the state of your resource, then the `owner` property calls the `owner` method on the provider. In turn, when you are setting the state (because the resource is out of sync), then the `owner` property calls the `owner=` method to set the state on disk.

There's one common exception to this: The `ensure` property is special because it's used to create and destroy resources. You can set this property up on your resource type just by calling the `ensurable` method in your type definition:

```
Puppet::Type.newtype(:database) do
  ensurable
  ...
end
```

This property uses three methods on the provider: `create`, `destroy`, and `exists?`. The last method, somewhat obviously, is a Boolean to determine if the resource exists. If a resource's `ensure` property is out of sync, then no other properties are checked or modified.

You can modify how `ensure` behaves, such as by adding other valid values and determining what methods get called as a result; see types like `package` for examples.

The rest of the properties are defined a lot like you define the types, with the `newproperty` method, which should be called on the type:

```
Puppet::Type.newtype(:database) do
  ensurable
  newproperty(:owner) do
    desc "The owner of the database."
    ...
  end
end
```

Note the call to `desc`; this sets the documentation string for this property, and for Puppet types that get distributed with Puppet, it is extracted as part of the Type reference.

When Puppet was first developed, there would typically be a lot of code in this property definition. Now, however, you only define valid values or set up validation and munging. If you specify valid values, then Puppet only accepts those values, and automatically handles accepting either strings or symbols. In most cases, you only define allowed values for `ensure`, but it works for other properties, too:

```
newproperty(:enable) do
  newvalue(:true)
  newvalue(:false)
end
```

You can attach code to the value definitions (this code would be called instead of the `property=` method), but it's normally unnecessary.

For most properties, though, it is sufficient to set up validation:

```
newproperty(:owner) do
  validate do |value|
    unless value =~ /^\\w+/
      raise ArgumentError, "%s is not a valid user name" % value
    end
  end
end
```

Note that the order in which you define your properties can be important: Puppet keeps track of the definition order, and it always checks and fixes properties in the order they are defined.

Customizing behavior

By default, if a property is assigned multiple values in an array:

- It is considered in sync if any of those values matches the current value.
- If none of those values match, the first one is used when syncing the property.

If, instead, the property should only be in sync if all values match the current value (for example, a list of times in a cron job), declare this:

```
newproperty(:minute, :array_matching => :all) do # :array_matching defaults
  to :first
  ...
end
```

You can also customize how information about your property gets logged. You can create an `is_to_s` method to change how the current values are described, `should_to_s` to change how the desired values are logged, and `change_to_s` to change the overall log message for changes. See current types for examples.

Handling property values

When a resource is created with a list of desired values, those values are stored in each property in its `@should` instance variable. You can retrieve those values directly by calling `should` on your resource (although note that when `:array_matching` is set to `:first` you get the first value in the array; otherwise you get the whole array):

```
myval = should(:color)
```

When you're not sure (or don't care) whether you're dealing with a property or parameter, it's best to use `value`:

```
myvalue = value(:color)
```

Parameters

Parameters are defined the same way as properties. The difference between them is that parameters never result in methods being called on providers.

To define a new parameter, call the `newparam` method. This method takes the name of the parameter (as a symbol) as its argument, as well as a block of code. You can and should provide documentation for each parameter by calling the `desc` method inside its block. Tools that generate docs from this description trim leading whitespace from multiline strings, as described for type descriptions.

```
newparam(:name) do
  desc "The name of the database."
end
```

Namevar

Every type must have at least one mandatory parameter: the `namevar`. This parameter uniquely identifies each resource of the type on the target system — for example, the path of a file on disk, the name of a user account, or the name of a package.

If the user doesn't specify a value for the namevar when declaring a resource, its value defaults to the title of the resource.

There are three ways to designate a namevar. Every type must have exactly one parameter that meets exactly one of these criteria:

1. Create a parameter whose name is `:name`. Because most types just use `:name` as the namevar, it gets special treatment and automatically becomes the namevar.

```
newparam(:name) do
  desc "The name of the database."
end
```

2. Provide the `:namevar => true` option as an additional argument to the `newparam` call. This allows you to use a namevar with a different, more descriptive name, such as the `file` type's `path` parameter.

```
newparam(:path, :namevar => true) do
  ...
end
```

3. Call the `isnamevar` method (which takes no arguments) inside the parameter's code block. This allows you to use a namevar with a different, more descriptive name. There is no practical difference between this and option 2.

```
newparam(:path) do
  isnamevar
  ...
end
```

Specifying allowed values

If your parameter has a fixed list of valid values, you can declare them all at the same time:

```
newparam(:color) do
  newvalues(:red, :green, :blue, :purple)
end
```

You can specify regular expressions in addition to literal values; matches against regex always happen after equality comparisons against literal values, and those matches are not converted to symbols. For instance, given the following definition:

```
newparam(:color) do
  desc "Your color, and stuff."

  newvalues(:blue, :red, /.+/)
end
```

If you provide `blue` as the value, then your parameter is set to `:blue`, but if you provide `green`, then it is set to `"green"`.

Validation and munging

If your parameter does not have a defined list of values, or you need to convert the values in some way, you can use the `validate` and `munge` hooks:

```
newparam(:color) do
```

```

desc "Your color, and stuff."

newvalues(:blue, :red, /.+/)

validate do |value|
  if value == "green"
    raise ArgumentError,
      "Everyone knows green databases don't have enough RAM"
  else
    super(value)
  end
end

munge do |value|
  case value
  when :mauve, :violet # are these colors really any different?
    :purple
  else
    super(value)
  end
end
end

```

The default `validate` method looks for values defined using `newvalues` and if there are any values defined it accepts only those values (this is how allowed values are validated). The default `munge` method converts any values that are specifically allowed into symbols. If you override either of these methods, note that you lose this value handling and symbol conversion, which you'll have to call `super` for.

Values are always validated before they're munged.

Lastly, validation and munging only happen when a value is assigned. They have no role to play at all during use of a given value, only during assignment.

Boolean parameters

Boolean parameters are common. To avoid repetition, some utilities are available:

```

require 'puppet/parameter/boolean'
# ...
newparam(:force, :boolean => true, :parent => Puppet::Parameter::Boolean)

```

There are two parts here. The `:parent => Puppet::Parameter::Boolean` part configures the parameter to accept lots of names for true and false, to make things easy for your users. The `:boolean => true` creates a `boolean` method on the type class to return the value of the parameter. In this example, the method would be named `force?`.

Automatic relationships

Use automatic relationships to define the ordering of resources.

By default, Puppet includes and processes resources in the order they are defined in their manifest. However, there are times when resources need to be applied in a different order. The Puppet language provides ways to express explicit ordering such as relationship metaparameters (`require`, `before`, etc), chaining arrows and the `require` and `contain` functions.

Sometimes there is natural relationship between your custom type and other resource types. For example, `ssh` authorized keys can only be managed after you create the home directory and you can only manage files after you create their parent directories. You can add explicit relationships for these, but doing so can be restrictive for others who may want to use your custom type. Automatic relationships provide a way to define implicit ordering. For example, to automatically add a `require` relationship from your custom type to a configuration file that it depends on, add the following to your custom type:

```

autorequire(:file) do

```

```
[ '/path/to/file' ]
end
```

The Ruby symbol `:file` refers to the type of resource you want to require, and the array contains resource title(s) with which to create the require relationship(s). The effect is nearly equivalent to using an explicit require relationship:

```
custom { <CUSTOM_RESOURCE>:
  ensure => present,
  require => File[ '/path/to/file' ]
}
```

An important difference between automatic and explicit relationships is that automatic relationships do not require the other resources to exist, while explicit relationships do.

Agent-side pre-run resource validation

A resource can have prerequisites on the target, without which it cannot be synced. In some cases, if the absence of these prerequisites would be catastrophic, you might want to halt the catalog run if you detect a missing prerequisite.

In this situation, define a method in your type named `pre_run_check`. This method can do any check you want. It should take no arguments, and should raise a `Puppet::Error` if the catalog run should be halted.

If a type has a `pre_run_check` method, Puppet agent and `puppet apply` runs the check for every resource of the type before attempting to apply the catalog. It collects any errors raised, and presents all of them before halting the catalog run.

As a trivial example, here's a pre-run check that fails randomly, about one time out of six:

```
Puppet::Type.newtype(:thing) do
  newparam :name, :namevar => true

  def pre_run_check
    if(rand(6) == 0)
      raise Puppet::Error, "Puppet roulette failed, no catalog for you!"
    end
  end
end
```

How types and providers interact

The type definition declares the features that a provider must have and what's required to make them work. Providers can either be tested for whether they suffice, or they can declare that they have the features. Because a type's properties call getter and setter methods on the providers, the providers must define getters and setters for each property (except `ensure`).

Additionally, individual properties and parameters in the type can declare that they require one or more specific features, and Puppet throws an error if those parameters are used with providers missing those features:

```
newtype(:coloring) do
  feature :paint, "The ability to paint.", :methods => [:paint]
  feature :draw, "The ability to draw."

  newparam(:color, :required_features => %w{paint}) do
    ...
  end
end
```

The first argument to the feature method is the name of the feature, the second argument is its description, and after that is a hash of options that help Puppet determine whether the feature is available. The only option currently

supported is specifying one or more methods that must be defined on the provider. If no methods are specified, then the provider needs to specifically declare that it has that feature:

```
Puppet::Type.type(:coloring).provide(:drawer) do
  has_feature :draw
end
```

The provider can specify multiple available features at the same time with `has_features`.

When you define features on your type, Puppet automatically defines the following class methods on the provider:

- `feature?:` Passed a feature name, returns true if the feature is available or false otherwise.
- `features:` Returns a list of all supported features on the provider.
- `satisfies?:` Passed a list of feature, returns true if they are all available, false otherwise.

Additionally, each feature gets a separate Boolean method, so the above example would result in a `paint?` method on the provider.

Related information

[Provider development](#) on page 480

Providers are back-ends that support specific implementations of a given resource type, particularly for different platforms. Not all resource types have or need providers, but any resource type concerned about portability will likely need them.

Provider development

Providers are back-ends that support specific implementations of a given resource type, particularly for different platforms. Not all resource types have or need providers, but any resource type concerned about portability will likely need them.

For instance, there are more than 20 package providers, including providers for package formats like `dpkg` and `rpm` along with high-level package managers like `apt` and `yum`. A provider's main job is to wrap client-side tools, usually by just calling out to those tools with the right information.

The examples on this page use the `apt` and `dpkg` package providers, and the examples used are current as of 0.23.0.

Note: Unless you are maintaining existing type and provider code, or the Resource API limitations affect you, use the Resource API to create custom resource types, instead of this method.

Declaring providers

Providers are always associated with a single resource type, so they are created by calling the `provide` method on that resource type.

The `provide` method takes three arguments plus a block:

- The first argument must be the name of the provider, as a `:symbol`.
- The optional `:parent` argument should be the name of a parent class.
- The optional `:source` argument should be a symbol.
- The block takes no arguments, and implements the behavior of the provider.

There are several kinds of parent classes you can use:

Base provider

A provider can inherit from a base provider, which is never used alone and only exists for other providers to inherit from. Use the full name of the class. For example, all package providers have a common parent class:

```
Puppet::Type.type(:package).provide(:dpkg, :parent =>
  Puppet::Provider::Package) do
  desc "..."


```
 ...
end
```


```


Note the call to the `desc` method; this sets the documentation for this provider, and should include everything necessary for someone to use this provider.

Another provider of the same resource type

Providers can also specify another provider as their parent. If it's a provider for the same resource type, you can use the name of that provider as a symbol.

```
Puppet::Type.type(:package).provide(:apt, :parent => :dpkg, :source
=> :dpkg) do
  ...
end
```

Note that we're also specifying that this provider uses the `dpkg` source; this tells Puppet to deduplicate packages from `dpkg` and `apt`, so the same package does not show up in an instance list from each provider type. Puppet defaults to creating a new source for each provider type, so you have to specify when a provider subclass shares a source with its parent class.

A provider of any resource type

Providers can also specify a provider of any resource type as their parent. Use the `Puppet::Type.type(<NAME>).provider(<NAME>)` methods to locate the provider.

```
# my_module/lib/puppet/provider/glance_api_config/ini_setting.rb
Puppet::Type.type(:glance_api_config).provide(
  :ini_setting,
  # set ini_setting as the parent provider
  :parent => Puppet::Type.type(:ini_setting).provider(:ruby)
) do
  # implement section as the first part of the namevar
  def section
    resource[:name].split('/', 2).first
  end
  def setting
    # implement setting as the second part of the namevar
    resource[:name].split('/', 2).last
  end
  # hard code the file path (this allows purging)
  def self.file_path
    '/etc/glance/glance-api.conf'
  end
end
```

Suitability

The first question to ask about a new provider is where it will be functional, which Puppet calls *suitable*. Unsuitable providers cannot be used to do any work. The suitability test is late-binding, meaning that you can have a resource in your configuration that makes a provider suitable.

If you start `puppetd` or `puppet` in debug mode, you'll see the results of failed provider suitability tests for the resource types you're using.

Puppet providers include some helpful class-level methods you can use to both document and declare how to determine whether a given provider is suitable. The primary method is `commands`, which does two things for you: it declares that this provider requires the named binary, and it sets up class and instance methods with the name provided that call the specified binary. The binary can be fully qualified, in which case that specific path is required, or it can be unqualified, in which case Puppet finds the binary in the shell path and uses that. If the binary cannot be found, then the provider is considered unsuitable. For example, here is the header for the `dpkg` provider (as of 0.23.0):

```
commands :dpkg => "/usr/bin/dpkg"
commands :dpkg_deb => "/usr/bin/dpkg-deb"
commands :dpkgquery => "/usr/bin/dpkg-query"
```

In addition to looking for binaries, Puppet can compare `Facter` facts, test for the existence of a file, check for a feature such as a library, or test whether a given value is true or false. For file existence, `true`, or `false`, call the `confine` class method with `exists`, `true`, or `false` as the name of the test and your test as the value:

```
confine :exists => "/etc/debian_release"
confine :true  => /^10\.[0-4]/.match(product_version)
confine :false => (Puppet[:ldapuser] == "")
```

To test `Facter` values, use the name of the fact:

```
confine :operatingsystem => [:debian, :centos]
confine :puppetversion  => "0.23.0"
```

Case doesn't matter in the tests, nor does it matter whether the values are strings or symbols. It also doesn't matter whether you specify an array or a single value — Puppet does an OR on the list of values.

To test a feature, as defined in `lib/puppet/feature/*.rb`, supply the name of the feature. This is preferable to using a `confine :true` statement that calls `Puppet.features` because the expression is evaluated only one time. Puppet enables the provider if the feature becomes available during a run (for example, if a package is installed during the run).

```
confine :feature => :posix
confine :feature => :rrd
```

You can create custom features. They live in `lib/puppet/feature/*.rb` and an example can be found [here](#). These features can be shipped in a similar manner as types and providers are shipped within modules and are `pluginsynced`.

Using custom features you can delay resource evaluation until the provider becomes suitable. This is a way of informing Puppet that your provider depends on a file being created by Puppet, or a certain fact being set to some value, or it not being set at all.

Default providers

Providers are generally meant to be hidden from the users, allowing them to focus on resource specification rather than implementation details, so Puppet does what it can to choose an appropriate default provider for each resource type.

This is generally done by a single provider declaring that it is the default for a given set of facts, using the `defaultfor` class method. For instance, this is the `apt` provider's declaration:

```
defaultfor :operatingsystem => :debian
```

The same fact-matching functionality as confinement is used.

Alternatively, you can supply a regular expression (regex) value to match against a fact value. This is useful, for example, for providers that should only be default for a specific range of operating system versions:

```
defaultfor :operatingsystemmajrelease => /^[5-7]$/
```

How providers interact with resources

Providers do nothing on their own; all of their action is triggered through an associated resource (or, in special cases, from the transaction). Because of this, resource types are essentially free to define their own provider interface if necessary, and providers were initially developed without a clear resource-provider API (mostly because it wasn't clear whether such an API was necessary or what it would look like). At this point, however, there is a default interface between the resource type and the provider.

This interface consists entirely of getter and setter methods. When the resource is retrieving its current state, it iterates across all of its properties and calls the getter method on the provider for that property. For instance, when a `user` resource is having its state retrieved and its `uid` and `shell` properties are being managed, then the resource calls

`uid` and `shell` on the provider to figure out what the current state of each of those properties is. This method call is in the `retrieve` method in `Puppet::Property`.

When a resource is being modified, it calls the equivalent setter method for each property on the provider. Again using our user example, if the `uid` was in sync but the `shell` was not, then the resource would call `shell=(value)` with the new shell value.

The transaction is responsible for storing these returned values and deciding which value to send, and it does its work through a `PropertyChange` instance. It calls `sync` on each of the properties, which in turn call the setter by default.

You can override that interface as necessary for your resource type.

All providers must define an `instances` class method that returns a list of provider instances, one for each existing instance of that provider. For example, the `dpkg` provider should return a provider instance for every package in the `dpkg` database.

Provider methods

By default, you have to define all of your getter and setter methods. For simple cases, this is sufficient — you just implement the code that does the work for that property. For the more complicated aspects of provider implementation, Puppet has prefetching, resource methods, and flushing.

Prefetching

First, Puppet transactions prefetch provider information by calling `prefetch` on each used provider type. This calls the `instances` method in turn, which returns a list of provider instances with the current resource state already retrieved and stored in a `@property_hash` instance variable. The `prefetch` method then tries to find any matching resources, and assigns the retrieved providers to found resources. This way you can get information on all of the resources you're managing in just a few method calls, instead of having to call all of the getter methods for every property being managed. Note that it also means that providers are often getting replaced, so you cannot maintain state in a provider.

Resource methods

For providers that directly modify the system when a setter method is called, there's no substitute for defining them manually. But for resources that get flushed to disk in one step, such as the `ParsedFile` providers, there is a `mk_resource_methods` class method that creates a getter and setter for each property on the resource. These methods retrieve and set the appropriate value in the `@property_hash` variable.

Flushing

Many providers model files or parts of files, so it makes sense to save up all of the writes and do them in one run. Providers that need this functionality can define a `flush` instance method to do this. The transaction calls this method after all values are synced (which means that the provider should have them all in its `@property_hash` variable) but before `refresh` is called on the resource (if appropriate).

Custom functions

Use the Puppet language, or the Ruby API to create custom functions.

For built-in Puppet functions, see the [Functions reference](#).

- [Custom functions overview](#) on page 484

Puppet includes many built-in functions, and more are available in modules on the Forge. You can also write your own custom functions.

- [Writing custom functions in the Puppet language](#) on page 485

You can write simple custom functions in the Puppet language, to transform data and construct values. A function can optionally take one or more parameters as arguments. A function returns a calculated value from its final expression.

- [Writing custom functions in Ruby](#) on page 489

You can write powerful and flexible functions using Ruby.

- [Deferring a function](#) on page 501

Deferring a function allows you to run code on the agent during a Puppet run.

Custom functions overview

Puppet includes many built-in functions, and more are available in modules on the Forge. You can also write your own custom functions.

Functions are plugins used during catalog compilation. When a Puppet manifest calls a function, that function runs and returns a value. Most functions only produce values, but functions can also:

- Cause side effects that modify a catalog. For example, the `include` function adds classes to a catalog.
- Evaluate a provided block of Puppet code, using arguments to determine how that code runs.

Functions usually take one or more arguments, which determine the return value and the behavior of any side effects.

If you need to manipulate data or communicate with third-party services during catalog compilation, and if the built-in functions, or functions from Forge modules, aren't sufficient, you can write custom functions for Puppet.

Custom functions work just like Puppet's built-in functions: you can call them during catalog compilation to produce a value or cause side effects. You can use your custom functions locally, and you can share them with other users.

To make a custom function available to Puppet, you must put it in a module or an environment, in the specific locations where Puppet expects to find functions.

Puppet offers two interfaces for writing custom functions:

Interface	Description
The Puppet language	To write functions in the Puppet language, you don't need to know any Ruby. However, it's less powerful than the Ruby API. Puppet functions can have only one signature per function, and can't take a lambda (a block of Puppet code) as an argument.
The Ruby functions API	The more powerful and flexible way to write functions. This method requires some knowledge of Ruby. You can use Ruby to write iterative functions.

Guidelines for writing custom functions

Whenever possible, avoid causing side effects. Side effects are any change other than producing a value, such as modifying the catalog by adding classes or resources to it.

In Ruby functions, it's possible to change the values of existing variables. Never do this, because Puppet relies on those variables staying the same.

Documenting functions

For information about documenting your functions, see [Puppet Strings](#).

Related topics: [Function calls](#), [Catalog compilation](#), [Environments](#), [Modules](#), [Puppet Forge](#), [Iterative functions](#).

Writing custom functions in the Puppet language

You can write simple custom functions in the Puppet language, to transform data and construct values. A function can optionally take one or more parameters as arguments. A function returns a calculated value from its final expression.

Note: While many functions can be written in the Puppet language, it doesn't support all of the same features as pure Ruby. For information about writing Ruby functions, which can perform more complex work, see [Writing functions in Ruby](#). For information about iterative functions, which can be invoked by, but not written exclusively with, Puppet code, see [Writing iterative functions](#).

Syntax of functions

```
function <MODULE NAME>::<NAME>(<PARAMETER LIST>) >> <RETURN TYPE> {
  ... body of function ...
  final expression, which is the returned value of the function
}
```

The general form of a function written in Puppet language is:

- The keyword `function`.
- The namespace of the function. This must match the name of the module the function is contained in.
- The namespace separator, a double colon `::`
- The name of the function.
- An optional parameter list, which consists of:
 - An opening parenthesis `(`
 - A comma-separated list of parameters (for example, `String $myparam = "default value"`). Each parameter consists of:
 - An optional data type, which restricts the allowed values for the parameter (defaults to `Any`).
 - A variable name to represent the parameter, including the `$` prefix.
 - An optional equals sign `=` and default value, which must match the data type, if one was specified.
 - An optional trailing comma after the last parameter.
 - A closing parenthesis `)`
- An optional return type, which consists of:
 - Two greater-than signs `>>`
 - A data type that matches every value the function could return.
- An opening curly brace `{`
- A block of Puppet code, ending with an expression whose value is returned.
- A closing curly brace `}`

For example:

```
function apache::bool2http(Variant[String, Boolean] $arg) >> String {
  case $arg {
    false, undef, /(?!i:false)/ : { 'Off' }
    true, /(?!i:true)/          : { 'On' }
    default                     : { "$arg" }
  }
}
```

Order and optional parameters

Puppet passes arguments by parameter position. This means that the order of parameters is important. Parameter names do not affect the order in which they are passed.

If a parameter has a default value, then it's optional to pass a value for it when you're calling the function. If the caller doesn't pass in an argument for that parameter, the function uses the default value. However, because parameters are

passed by position, when you write the function, you must list optional parameters after all required parameters. If you put a required parameter after an optional one, it causes an evaluation error.

Variables in default parameters values

If you reference a variable as a default value for a parameter, Puppet starts looking for that variable at top scope. For example, if you use `$fqdn` as a variable, but then call the function from a class that overrides `$fqdn`, the parameter's default value is the value from top scope, not the value from the class. You can reference qualified variable names in a function default value, but compilation fails if that class isn't declared by the time the function is called.

The extra arguments parameter

You can specify that a function's last parameter is an extra arguments parameter. The extra arguments parameter collects an unlimited number of extra arguments into an array. This is useful when you don't know in advance how many arguments the caller provides.

To specify that the parameter must collect extra arguments, start its name with an asterisk `*`, for example `*$others`. The asterisk is valid only for the last parameter.

Tip: An extra argument's parameter is always optional.

The value of an extra argument's parameter is an array, containing every argument in excess of the earlier parameters. You can give it a default value, which has some automatic array wrapping for convenience:

- If the provided default is a non-array value, the real default is a single-element array containing that value.
- If the provided default is an array, the real default is that array.

If there are no extra arguments and there is no default value, it's an empty array.

An extra arguments parameter can also have a data type. Puppet uses this data type to validate the elements of the array. That is, if you specify a data type of `String`, the real data type of the extra arguments parameter is `Array[String]`.

Return types

Between the parameter list and the function body, you can use `>>` and a data type to specify the type of the values the function returns. For example, this function only returns strings:

```
function apache::bool2http(Variant[String, Boolean] $arg) >> String {
  ...
}
```

The return type serves two purposes:

- Documentation. Puppet Strings includes information about the return value of a function.
- Insurance. If something goes wrong and your function returns the wrong type (such as `undef` when a string is expected), it fails early with an informative error instead of allowing compilation to continue with an incorrect value.

The function body

In the function body, put the code required to compute the return value you want, given the arguments passed in. Avoid declaring resources in the body of your function. If you want to create resources based on inputs, use [defined types](#) instead.

The final expression in the function body determines the value that the function returns when called. Most conditional expressions in the Puppet language have values that work in a similar way, so you can use an `if` statement or a

case statement as the final expression to give different values based on different numbers or types of inputs. In the following example, the case statement serves as both the body of the function, and its final expression.

```
function apache::bool2http(Variant[String, Boolean] $arg) >> String {
  case $arg {
    false, undef, /(?:false)/ : { 'Off' }
    true, /(?:true)/          : { 'On' }
    default                    : { "$arg" }
  }
}
```

Locations

Store the functions you write in a module's `functions` folder, which is a top-level directory (a sibling of `manifests` and `lib`). Define only one function per file, and name the file to match the name of the function being defined. Puppet is automatically aware of functions in a valid module and autoloads them by name.

Avoid storing functions in the main manifest. Functions in the main manifest override any function of the same name in all modules (except built-in functions).

Names

Give your function a name that clearly reveals what it does. For more information about names, including restrictions and reserved words, see [Puppet naming conventions](#).

Related topics: [Arrays](#), [Classes](#), [Data types](#), [Conditional expressions](#), [Defined types](#), [Namespaces and autoloading](#), [Variables](#).

Calling a function

A call to a custom function behaves the same as a call to any built-in Puppet function, and resolves to the function's returned value.

After a function is written and available in a module where the autoloader can find it, you can call that function, either from a Puppet manifest that lists the containing module as a dependency, or from your main manifest.

Any arguments you pass to the function are mapped to the parameters defined in the function's definition. You must pass arguments for the mandatory parameters, but you can choose whether you want to pass in arguments for optional parameters.

Functions are autoloaded and are available to other modules unless those modules have specified dependencies. If a module has a list of dependencies in its `metadata.json` file, only custom functions from those specific dependencies are loaded.

Related topics: [namespaces and autoloading](#), [module metadata](#), [main manifest directory](#)

Complex example of a function

The following code example is a re-written version of a Ruby function from the `postgresql` module into Puppet code. This function translates the IPv4 and IPv6 Access Control Lists (ACLs) format into a resource suitable for `create_resources`. In this case, the filename would be `acls_to_resource_hash.pp`, and it would be saved in a folder named `functions` in the top-level directory of the `postgresql` module.

```
function postgresql::acls_to_resource_hash(Array $acls, String $id, Integer
  $offset) {

  $func_name = "postgresql::acls_to_resources_hash()"

  # The final hash is constructed as an array of individual hashes
  # (using the map function), the result of that
  # gets merged at the end (using reduce).
```

```

#
$resources = $acl.smap |$index, $acl| {
  $parts = $acl.split('\s+')
  unless $parts =~ Array[Data, 4] {
    fail("${func_name}: acl line $index does not have enough parts")
  }

  # build each entry in the final hash
  $resource = { "postgresql class generated rule ${id} ${index}" =>
    # The first part is the same for all entries
    {
      'type'      => $parts[0],
      'database' => $parts[1],
      'user'      => $parts[2],
      'order'     => sprintf("%03d", $offset + $index)
    }
    # The rest depends on if first part is 'local',
    # the length of the parts, and the value in $parts[4].
    # Using a deep matching case expression is a good way
    # to untangle if-then-else spaghetti.
    #
    # The conditional part is merged with the common part
    # using '+' and the case expression results in a hash
    #
    +
    case [$parts[0], $parts, $parts[4]] {

      ['local', Array[Data, 5], default] : {
        { 'auth_method' => $parts[3],
          'auth_option' => $parts[4, -1].join(" ")
        }
      }

      ['local', default, default] : {
        { 'auth_method' => $parts[3] }
      }

      [default, Array[Data, 7], /^d/] : {
        { 'address'      => "${parts[3]} ${parts[4]}",
          'auth_method' => $parts[5],
          'auth_option' => $parts[6, -1].join(" ")
        }
      }

      [default, default, /^d/] : {
        { 'address'      => "${parts[3]} ${parts[4]}",
          'auth_method' => $parts[5]
        }
      }

      [default, Array[Data, 6], default] : {
        { 'address'      => $parts[3],
          'auth_method' => $parts[4],
          'auth_option' => $parts[5, -1].join(" ")
        }
      }

      [default, default, default] : {
        { 'address'      => $parts[3],
          'auth_method' => $parts[4]
        }
      }
    }
  }
}

```



```

    $resource
  }
  # Merge the individual resource hashes into one
  $resources.reduce({}) |$result, $resource| { $result + $resource }
}

```

Writing custom functions in Ruby

You can write powerful and flexible functions using Ruby.

- [Custom functions in Ruby overview](#) on page 489

Get started with an overview of Ruby custom functions.

- [Ruby function signatures](#) on page 491

Functions can specify how many arguments they expect, and a data type for each argument. The rule set for a function's arguments is called a signature.

- [Using special features in implementation methods](#) on page 495

For the most part, implementation methods are normal Ruby. However, there are some special features available for accessing Puppet variables, working with provided blocks of Puppet code, and calling other functions.

- [Iterative functions](#) on page 497

You can use iterative types to write efficient iterative functions, or to chain together the iterative functions built into Puppet.

- [Refactoring legacy 3.x functions](#) on page 499

If you have Ruby functions written with the legacy 3.x API, refactor them to ensure that they work correctly with current versions of Puppet.

Custom functions in Ruby overview

Get started with an overview of Ruby custom functions.

Syntax of Ruby functions

To write a new function in Ruby, use the `Puppet::Functions.create_function` method. You don't need to require any Puppet libraries. Puppet handle libraries automatically when it loads the function.

```

Puppet::Functions.create_function(:<FUNCTION NAME>) do
  dispatch :<METHOD NAME> do
    param '<DATA TYPE>', :<ARGUMENT NAME (displayed in docs/errors)>
    ...
  end

  def <METHOD NAME>(<ARGUMENT NAME (for local use)>, ...)
    <IMPLEMENTATION>
  end
end

```

The `create_function` method requires:

- A function name.
- A block of code (which takes no arguments). This block contains:
 - One or more [signatures](#) to configure the function's arguments.
 - An implementation method for each signature. The return value of the implementation method is the return value of the function.

For example:

```

# /etc/puppetlabs/code/environments/production/modules/mymodule/lib/puppet/
functions/mymodule/upcase.rb
Puppet::Functions.create_function(:'mymodule::upcase') do
  dispatch :up do
    param 'String', :some_string
  end
end

```

```
def up(some_string)
  some_string.upcase
end
end
```

Location

Place a Ruby function in its own file, in the `lib/puppet/functions` directory of either a module or an environment. The filename must match the name of the function, and have the `.rb` extension. For namespaced functions, each segment prior to the final one must be a subdirectory of `functions`, and the final segment must be the filename.

Function name	File location
<code>upcase</code>	<code><MODULES DIR>/mymodule/lib/puppet/functions/upcase.rb</code>
<code>upcase</code>	<code>/etc/puppetlabs/code/environments/production/lib/puppet/functions/upcase.rb</code>
<code>mymodule::upcase</code>	<code><MODULES DIR>/mymodule/lib/puppet/functions/mymodule/upcase.rb</code>
<code>environment::upcase</code>	<code>/etc/puppetlabs/code/environments/production/lib/puppet/functions/environment/upcase.rb</code>

Functions are autoloaded and made available to other modules unless those modules specify dependencies. After a function is written and available (in a module where the autoloader can find it), you can call that function in any Puppet manifest that lists the containing module as a dependency, and also from your main manifest. If a module has a list of dependencies in its `metadata.json` file, it loads custom functions only from those specific dependencies.

Names

Function names are similar to class names. They consist of one or more segments. Each segment must start with a lowercase letter, and can include:

- Lowercase letters
- Numbers
- Underscores

If a name has multiple segments, separate them with a double-colon (`::`) namespace separator.

Match each segment with this regular expression:

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

Match the full name with this regular expression:

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

Function names can be either global or namespaced:

- Global names have only one segment (`str2bool`), and can be used in any module or environment. Global names are shorter, but they're not guaranteed to be unique — if you use a function name that is already in use by another module, Puppet might load the wrong module when you call it.
- Namespaced names have multiple segments (`stdlib::str2bool`), and are guaranteed to be unique. The first segment is dictated by the function's location:
 - In an environment, use `environment` (`environment::str2bool`).
 - In a module, use the module's name (`stdlib::str2bool` for a function stored in the `stdlib` module).

Most functions have two name segments, although it's legal to use more.

Examples of legal function names:

- `num2bool` (a function that could come from anywhere)
- `postgresql::acls_to_resource_hash` (a function in the `postgresql` module)
- `environment::hash_from_api_call` (a function in an environment)

Examples of illegal function names:

- `6_pack` (must start with a letter)
- `_hash_from_api_call` (must start with a letter)
- `Find-Resource` (can only contain lowercase letters, numbers, and underscores)

Passing names to `create_function` as symbols

When you call the `Puppet::Functions.create_function` method, pass the function's name to it as a Ruby symbol.

To turn a function name into a symbol:

- If the name is global, prefix it with a colon (`:str2bool`).
- If it's namespaced: put the name in quotation marks, and prefix the full quoted string with a colon (`:'stdlib::str2bool'`).

Related topics: [Puppet modules](#), [Environments](#), [Main manifest](#), [Module metadata](#), [Ruby symbols](#).

Ruby function signatures

Functions can specify how many arguments they expect, and a data type for each argument. The rule set for a function's arguments is called a signature.

Because Puppet functions support more advanced argument checking than Ruby does, the Ruby functions API uses a lightweight domain-specific language (DSL) to specify signatures.

Ruby functions can have multiple signatures. Using multiple signatures is an easy way to have a function behave differently when passed by different types or quantities of arguments. Instead of writing complex logic to decide what to do, you can write separate implementations and let Puppet select the correct signature.

If a function has multiple signatures, Puppet uses its data type system to check each signature in order, comparing the allowed arguments to the arguments that were actually passed. As soon as Puppet finds a signature that can accept the provided arguments, it calls the associated implementation method, passing the arguments to that method. When the method finishes running and returns a value, Puppet uses that as the function's return value. If none of the function's signatures match the provided arguments, Puppet fails compilation and logs an error message describing the mismatch between the provided and expected arguments.

Conversion of Puppet and Ruby data types

When function arguments are passed to a Ruby method, they're converted to Ruby objects. Similarly, when the Puppet manifest regains control, it converts the method's return value into a Puppet data type.

Puppet converts data types between the Puppet language and Ruby as follows:

Puppet	Ruby
Boolean	Boolean
Undef	NilClass (value nil)
String	String
Number	subtype of Numeric
Array	Array
Hash	Hash

Puppet	Ruby
Default	Symbol (value :default)
Regex	Regex
Resource reference	Puppet::Pops::Types::PResourceType, or Puppet::Pops::Types::PHostClassType
Lambda (code block)	Puppet::Pops::Evaluator::Closure
Data type (Type)	A type class under Puppet::Pops::Types. For example, Puppet::Pops::Types::PIntegerType

Tip: When writing iterative functions, use [iterative types](#) instead of Puppet types.

Writing signatures with `dispatch`

To write a signature, use the `dispatch` method.

The `dispatch` method takes:

- The name of an implementation method, provided as a Ruby symbol. The corresponding method must be defined somewhere in the `create_function` block, usually after all the signatures.
- A block of code which only contains calls to the parameter and return methods.

```
# A signature that takes a single string argument
dispatch :camelcase do
  param 'String', :input_string
  return_type 'String' # optional
end
```

Using parameter methods

In the code block of a `dispatch` statement, you can specify arguments with special parameter methods. All of these methods take two arguments:

- The allowed data type for the argument, as a string. Types are specified using Puppet's data type syntax.
- A user-facing name for the argument, as a symbol. This name is only used in documentation and error messages; it doesn't have to match the argument names in the implementation method.

The order in which you call these methods is important: the function's first argument goes first, followed by the second, and so on. The following parameter methods are available:

Model name	Description
<code>param</code> or <code>required_param</code>	A mandatory argument. You can use any number of these. Position: All mandatory arguments must come first.
<code>optional_param</code>	An argument that can be omitted. You can use any number of these. When there are multiple optional arguments, users can only pass latter ones if they also provide values for the prior ones. This also applies to repeated arguments. Position: Must come after any required arguments.

Model name	Description
repeated_param or optional_repeated_param	<p>A repeatable argument, which can receive zero or more values. A signature can only use one repeatable argument.</p> <p>Position: Must come after any non-repeating arguments.</p>
required_repeated_param	<p>A repeatable argument, which must receive one or more values. A signature can only use one repeatable argument.</p> <p>Position: Must come after any non-repeating arguments.</p>
block_param or required_block_param	<p>A mandatory lambda (block of Puppet code). A signature can only use one block.</p> <p>Position: Must come after all other arguments.</p>
optional_block_param	<p>An optional lambda. A signature can only use one block.</p> <p>Position: Must come after all other arguments.</p>

When specifying a repeatable argument, note that:

- In your implementation method, the repeatable argument appears as an array, which contains all the provided values that weren't assigned to earlier, non-repeatable arguments.
- The specified data type is matched against each value for the repeatable argument, not the repeatable argument as a whole. For example, if you want to accept any number of numbers, specify `repeated_param 'Numeric', :values_to_average`, not `repeated_param 'Array[Numeric]', :values_to_average`.

For lambdas, note that:

- The data type for a block argument is `Callable`, or a `Variant` that only contains `Callables`.
- The `Callable` type can optionally specify the type and quantity of parameters that the lambda accepts. For example, `Callable[String, String]` matches any lambda that can be called with a pair of strings.

Matching arguments with implementation methods

The implementation method that corresponds to a signature must be able to accept any combination of arguments that the signature might allow.

If the signature has optional arguments, the corresponding method arguments need default values. Otherwise, the function fails if the arguments are omitted. For example:

```
dispatch :epp do
  required_param 'String', :template_file
  optional_param 'Hash', :parameters_hash
end

def epp(template_file, parameters_hash = {})
  # Note that parameters_hash defaults to an empty hash.
end
```

If the signature has a repeatable argument, the method must use a splat parameter (`*args`) as its final argument. For example:

```
dispatch :average do
  required_repeated_param 'Numeric', :values_to_average
end
```

```
def average(*values)
  # Inside the method, the `values` variable is an array of numbers.
end
```

Using the `return_type` method

After specifying a signature's arguments, you can use the `return_type` method to specify the data type of its return value. This method takes one argument: a Puppet data type, specified as a string.

```
dispatch :camelcase do
  param 'String', :input_string
  return_type 'String'
end
```

The return type serves two purposes: documentation, and insurance.

- Puppet Strings can include information about the return value of a function.
- If something goes wrong and your function returns the wrong type (like `nil` when a string is expected), it fails early with an informative error instead of allowing compilation to continue with an incorrect value.

Specifying aliases using `local_types`

If you're using complicated abstract data types to validate arguments, and you're using these data types in multiple signatures, they can become difficult to work with and maintain. In these cases, you can specify short aliases for your complex data types and use the aliases in your signatures.

To specify aliases, use the `local_types` method:

- You must call `local_types` only one time, before any signatures.
- The `local_types` method takes a lambda, which only contains calls to the `type` method.
- The `type` method takes a single string argument, in the form '`<NAME> = <TYPE>`'.
 - Capitalize the name, camel case word (`PartColor`), similar to a Ruby class name or the existing Puppet data types.
 - The type is a valid Puppet data type.

Example:

```
local_types do
  type 'PartColor = Enum[blue, red, green, mauve, teal, white, pine]'
  type 'Part = Enum[cubicle_wall, chair, wall, desk, carpet]'
  type 'PartToColorMap = Hash[Part, PartColor]'
end

dispatch :define_colors do
  param 'PartToColorMap', :part_color_map
end

def define_colors(part_color_map)
  # etc
end
```

Using automatic signatures

If your function only needs one signature, and you're willing to skip the API's data type checking, you can use an automatic signature. Be aware that there are some drawbacks to using automatic signatures.

Although functions with automatic signatures are simpler to write, they give worse error messages when called incorrectly. You'll get a useful error if you call the function with the wrong number of arguments, but if you give the wrong type of argument, you'll get something unhelpful. For example, if you pass the function above a

number instead of a string, it reports `Error: Evaluation Error: Error while evaluating a Function Call, undefined method 'split' for 5:Fixnum at /Users/nick/Desktop/test2.pp:7:8 on node magpie.lan.`

If it's possible that your function will be used by anyone other than yourself, support your users by writing a signature with `dispatch`.

To use an automatic signature:

- Do not write a `dispatch` block.
- Define one implementation method whose name matches the final namespace segment of the function's name.

```
Puppet::Functions.create_function(:'stdlib::camelcase') do
  def camelcase(str)
    str.split('_').map{|e| e.capitalize}.join
  end
end
```

In this case, because the last segment of `stdlib::camelcase` is `camelcase`, we must define a method named `camelcase`.

Related topics: [Ruby symbols](#), [Abstract data types](#).

Using special features in implementation methods

For the most part, implementation methods are normal Ruby. However, there are some special features available for accessing Puppet variables, working with provided blocks of Puppet code, and calling other functions.

Accessing Puppet variables

We recommend that most functions only use the arguments they are passed. However, you also have the option of accessing globally-reachable Puppet variables. The main use case for this is accessing facts, trusted data, or server data.

Remember: Functions cannot access local variables in the scope from which they were called. They can only access global variables or fully-qualified class variables.

To access variables, use the special `closure_scope` method, which takes no arguments and returns a `Puppet::Parser::Scope` object.

Use `#[] (varname)` to call on a scope object, which returns the value of the specified variable. Make sure to exclude the `$` from the variable name. For example:

```
Puppet::Functions.create_function(:'mymodule::fqdn_rand') do
  dispatch :fqdn do
    # no arguments
  end

  def fqdn()
    scope = closure_scope
    fqdn = scope['facts']['networking']['fqdn']
    # ...
  end
end
```

Working with lambdas (code blocks)

If their signatures allow it, functions can accept lambdas (blocks of Puppet code). If a function has a lambda, it generally needs to execute it. To do this, use Ruby's normal block calling conventions.

Convention	Description
<code>block_given?</code>	If your signature specified an optional code block, your implementation method can check for its presence with the <code>block_given?</code> method. This is <code>true</code> if a block was provided, <code>false</code> if not.
<code>yield()</code>	<p>When you know a block was provided, you can execute it any number of times with the <code>yield()</code> method.</p> <p>The arguments to <code>yield</code> are passed as arguments to the lambda. If your signature specified the number and type of arguments the lambda expects, you can call it with confidence. The return value of the <code>yield</code> call is the return value of the provided lambda.</p>

If you need to introspect a provided lambda, or pass it on to some other method, an implementation method can capture it as a `Proc` by specifying an extra argument with an ampersand (&) flag. This works the same way as capturing a Ruby block as a `Proc`. After you capture the block, you can execute it with `#call` instead of `yield`. You can also use any other `Proc` instance methods to examine it.

```
def implementation(arg1, arg2, *splat_arg, &block)
  # Now the `block` variable has the provided lambda, as a Proc.
  block.call(arg1, arg2, splat_arg)
end
```

Calling other functions

If you want to call another Puppet function (like `include`) from inside a function, use the special `call_function(name, *args, &block)` method.

```
# Flatten an array of arrays of strings, then pass it to include:
def include_nested(array_of_arrays)
  call_function('include', *array_of_arrays.flatten)
end
```

- The first argument must be the name of the function to call, as a string.
- The next arguments can be any data type that the called function accepts. They are passed as arguments to the called function.
- The last argument can be a Ruby `Proc`, or a Puppet lambda previously captured as a `Proc`. You can also provide a block of Ruby code using the normal block syntax.

```
def my_function1(a, b, &block)
  # passing given Proc
  call_function('my_other_function', a, b, &block)
end

def my_function2(a, b)
  # using a Ruby block
  call_function('my_other_function', a, b) { |x| ... }
end
```

Related topics: [Proc](#), [yield](#), [block_given?](#), [Puppet variables](#), [Lambdas](#), [Facts and built-in variables](#).

Iterative functions

You can use iterative types to write efficient iterative functions, or to chain together the iterative functions built into Puppet.

Iterative functions include `Iterable` and `Iterator` types, as well as other types you can iterate over, such as arrays and hashes. For example, an `Array[Integer]` is also an `Iterable[Integer]`.

Tip: `Iterable` and `Iterator` types are used internally by Puppet to efficiently chain the results of its built-in iterative functions. You can't write iterative functions solely in the Puppet language. For help writing less complex functions in Puppet code, see [Writing functions in Puppet](#).

Iterable and Iterator type design

The `Iterable` type represents all things an iterative function can iterate over. Before this type was introduced in Puppet 4.4, if you wanted to design working iterative functions, you'd have to write code that accommodated all relevant types, such as `Array`, `Hash`, `Integer`, and `Type[Integer]`.

Signatures of iterative functions accept an `Iterable` type argument. This means that you no longer have to design iterative functions to check against every type. This behavior does not affect how the Puppet code that invokes these functions works, but does change the errors you see if you try to iterate over a value that does not have the `Iterable` type.

The `Iterator` type, which is a subtype of `Iterable`, is a special algorithm-based `Iterable` not backed by a concrete data type. When asked to produce a value, an `Iterator` produces the next value from its input, and then either yields a transformation of this value, or takes its input and yields each value from a formula based on that value. For example, the `step` function produces consecutive values but does not need to first produce an array containing all of the values.

Writing iterative functions

Remember: You can't write iterative functions solely in the Puppet language.

When writing iterative functions, use the `Iterable` type instead of the more specific, individual types. The `Iterable` type has a type parameter that describes the type that is yielded in each iteration. For example, an `Array[Integer]` is also an `Iterable[Integer]`.

When writing a function that returns an `Iterator`, declare the return type as `Iterable`. This is the most flexible way to handle an `Iterator`.

For best practices on implementing iterative functions, [examine existing iterative functions in Puppet](#) and read the Ruby documentation for the helper classes these functions use. See the implementations of `each` and `map` for functions that always produce a new result, and `reverse_each` and `step` for new iterative functions that return an `Iterable` when called without a block.

For example, this is the Ruby code for the `step` function:

```
Puppet::Functions.create_function(:step) do
  dispatch :step do
    param 'Iterable', :iterable
    param 'Integer[1]', :step
  end

  dispatch :step_block do
    param 'Iterable', :iterable
    param 'Integer[1]', :step
    block_param 'Callable[1,1]', :block
  end

  def step(iterable, step)
    # produces an Iterable
    Puppet::Pops::Types::Iterable.asserted_iterable(self,
iterable).step(step)
  end
end
```

```

def step_block(iterable, step, &block)
  Puppet::Pops::Types::Iterable.asserted_iterable(self,
    iterable).step(step, &block)
  nil
end
end

```

Efficiently chaining iterative functions

Iterative functions are often used in chains, where the result of one function is used as the next function's parameter. A typical example is a map/reduce function, where values are first modified, and then an aggregate value is computed. For example, this use of `reverse_each` and `reduce`:

```
[1,2,3].reverse_each.reduce |$result, $x| { $result - $x }
```

The `reverse_each` function iterates over the `Array` to reverse the order of its values from `[1, 2, 3]` to `[3, 2, 1]`. The `reduce` function iterates over the `Array`, subtracting each value from the previous value. The `$result` is 0, because $3 - 2 - 1 = 0$.

Iterable types allow functions like these to execute more efficiently in a chain of calls, because they eliminate each function's need to create an intermediate copy of the mapped values in the appropriate type. In the above example, the mapped values would be the array `[3, 2, 1]` produced by the `reverse_each` function. The first time the `reduce` function is called, it receives the values 3 and 2 — the value 1 has not yet been computed. In the next iteration, `reduce` receives the value 1, and the chain ends because there are no more values in the array.

Limitations and workarounds

When you use it last in a chain, you can assign a value of `Iterator[T]` (where `T` is a data type) to a variable and pass it on. However, you cannot assign an `Iterator` to a parameter value. It's also not possible to call legacy 3.x functions with an `Iterator`.

If you assign an `Iterator` to a resource attribute, you get an error. This is because the `Iterator` type is a special algorithm-based `Iterable` that is not backed by a concrete data type. In addition, parameters in resources are serialized, and Puppet cannot serialize a temporary algorithmic result.

For example, if you used the following Puppet code:

```

notify { 'example1':
  message => [1,2,3].reverse_each,
}

```

You would receive the following error:

```
Error while evaluating a '=>' expression, Use of an Iterator is not supported here
```

Puppet needs a concrete data type for serialization, but the result of `[1, 2, 3].reverse_each` is only a temporary `Iterator` value. To convert the `Iterator`-typed value to an `Array`, map the value.

This example results in an array by chaining the `map` function:

```

notify { 'mapped_iterator':
  message => [1,2,3].reverse_each.map |$x| { $x },
}

```

You can also use the splat operator `*` to convert the value into an array.

```

notify { 'mapped_iterator':
  message => *[1,2,3].reverse_each,
}

```

```
}
```

Both of these examples result in a notice containing `[3 , 2 , 1]`. If you use `*` in a context where it also unfolds, the result is the same as unfolding an array: each value of the array becomes a separate value, which results in separate arguments in a function call.

Related topics: [step functions](#), [each functions](#), [reduce functions](#), [map functions](#), [reverse_each functions](#).

Refactoring legacy 3.x functions

If you have Ruby functions written with the legacy 3.x API, refactor them to ensure that they work correctly with current versions of Puppet.

Refactoring legacy functions improves functionality and prevents errors. At minimum, refactor any extra methods in your 3.x functions, because these no longer work in Puppet.

Extra methods

Legacy functions that contain methods defined inside the function body or outside of the function return an error, such as:

```
raise SecurityError, _("Illegal method definition of method '%{method_name}'
  on line %{line}' in legacy function") % { method_name: mname, line: mline }
```

To fix these errors, refactor your 3.x functions to the 4.x function API, where defining multiple methods is permitted.

For example, the legacy function below has been refactored into the modern API, with the following changes:

- Documentation for the function is now a comment before the call to `create_function`.
- The default dispatcher dispatches all given arguments to a method with the same name as the function.
- The `extra_method` has not been moved, but is legal in the modern API.
- Not visible in the code sample, the function has been moved from `lib/puppet/parser/functions` to `lib/puppet/functions`.

3.x API function:

```
module Puppet::Parser::Functions
  newfunction(:sample, :type => :rvalue, :doc => <<-EOS
    The function's documentation
  EOS
  ) do |arguments|
    "the returned value"
  end

  def extra_method()
    end
end
```

4.x API function:

```
# The function's documentation
Puppet::Functions.create_function(:sample) do

  def sample(*arguments)
    "the returned value"
  end

  def extra_method()
    end
end
```

Function call forms

Change all function calls from the form `function_***` to use the method `call_function(name, args)`.

The `function_***` form applies only to functions implemented in the 3.x API, so function with calls in that form can not call any function that has moved to the 4.x API.

For example, a 3.x function:

```
function_sprintf("%s", "example")
```

The refactored 4.x function:

```
call_function('sprintf', "%s", "example")
```

:rvalue specification

The 3.x API differentiated between functions returning a value (`:type => :rvalue`) and functions that did not return a value (`:type => :statement`). In the 4.x API, there is no such distinction. If you are refactoring a function where `:rvalue => true`, you do not need to make any changes. If you are refactoring a function where `:rvalue => false`, make sure the function returns `nil`.

Data values

The 4.x function API allows certain data values, such as `Regexp`, `Timespan`, and `Timestamp`. However, the 3.x API transformed these and similar data values into strings.

Review the logic in your refactored function with this in mind: instead of checking for empty strings, the function checks for `nil`. The function uses neither empty strings nor the `:undef` symbol in returned values to denote `undef`; again, use `nil` instead.

For `String`, `Integer`, `Float`, `Array`, `Hash`, and `Boolean` values, you do not need to make changes to your 3.x functions.

Documentation

The 4.x API supports Markdown and Puppet Strings documentation tags to document functions, including individual parameters and returned values. See the [Strings documentation](#) page for details about the correct format and style for documentation comments.

Namespacing

Namespace your function to the module in which it is defined, and update manifests that use it.

The function name is in the format `module_name::function_name`. For example, if the module name is `mymodule`:

```
# The function's documentation
Puppet::Functions.create_function(:'mymodule::sample') do

  def sample(*arguments)
    "the returned value"
  end

  def extra_method()
    end
end
```

The default dispatch uses the last part of the name when dispatching to a method in the function, so you only have to change the module namespace in the function's full name. You must also move the file containing the function to the correct location for 4.x API functions, `mymodule/lib/puppet/functions`.

Deferring a function

Deferring a function allows you to run code on the agent during a Puppet run.

- [Deferred functions overview](#) on page 501

Defer a function to retrieve a value on the agent.

- [Using a template with Deferred values](#) on page 502

Templates are rendered on the primary server during catalog compilation. However, this won't work with deferred functions because their values aren't known at compile time. Instead, you need to defer the template rendering.

- [Write a deferred function to store secrets](#) on page 503

Use the `Deferred` type to create a function that you add to a module to redact sensitive information.

- [Integrations with secret stores](#) on page 504

The Forge already hosts some community modules that provide integrations with secret stores.

Deferred functions overview

Defer a function to retrieve a value on the agent.

The `Deferred` type instructs agents to execute a function locally to retrieve a data value at the time of catalog application. When compiling catalogs, functions are normally executed on the primary server, with results entered into the catalog directly. The complete and fully resolved catalog is then sent to the agent for application. You can choose to defer the function call until the agent applies the catalog, meaning the agent calls the function on the agent instead of on the primary server. This way, agents can use a function to fetch data directly, rather than having the primary server act as an intermediary.

The two most common reasons for deferring functions are:

- To retrieve a value on the agent that the primary server doesn't have access to, including sensitive information like passwords from a secrets store.
- To use as a placeholder to describe intent and make your desired state as descriptive as possible.

Deferred function example

The following example shows a file with a template on the agent that defers two functions — the Vault password lookup and the `epp` template compilation.

The password lookup is deferred to run on the agent so that the primary server does not need to know the secret, and the `epp()` function is deferred to render the template after the password value is available.

```
$variables = {
  'password' => Deferred('vault_lookup::lookup',
                        ["secret/test", 'https://vault.docker:8200']),
}

# compile the template source into the catalog
file { ['/etc/secrets.conf':
  ensure => file,
  content => Deferred('inline_epp',
                    ['PASSWORD=<%= $password.unwrap %>', $variables]),
}
```

The `Deferred` object initialization signature returns an object that you can assign to a variable, pass to a function, or use like any other Puppet object. For example:

```
Deferred( <name of function to invoke>, [ array, of, arguments] )
```

This object compiles directly into the catalog and its function is invoked as the first part of enforcing a catalog. It is replaced by whatever it returns, similar to string interpolation. The catalog looks like the JSON hash below.

The password key is replaced with the results of the `vault_lookup::lookup` invocation, and then the content key is replaced with the results of the `inline_epp` invocation. Puppet can then manage the contents of the file without the primary server ever knowing the secret.

```
$ jq '.resources[] | select(.type == "File" and .title == "/etc/secrets.conf")' catalog.json
{
  "type": "File",
  "title": "/etc/secrets.conf",
  "parameters": {
    "ensure": "file",
    "owner": "root",
    "group": "root",
    "mode": "0600",
    "content": {
      "__ptype": "Deferred",
      "name": "inline_epp",
      "arguments": [
        "PASSWORD=$password\n",
        {
          "password": {
            "__ptype": "Deferred",
            "name": "vault_lookup::lookup",
            "arguments": ["secret/test", "https://vault.docker:8200"]
          }
        }
      ]
    },
    "backup": false
  }
}
```

Note:

When using deferred functions, take note of the following:

- If an agent is applying a cached catalog, the `Deferred` function is still called at application time, and the value returned at that time is the value that is used.
- It is the responsibility of the function to handle edge cases such as providing default or cached values in cases where a remote store is unavailable.
- `Deferred` supports only the Puppet function API for Ruby.
- If a function called on the agent side does not return `Sensitive`, you can wrap the value returned by `Deferred` in a `Sensitive` type if a sensitive value is desired. For example: `$d = Sensitive(Deferred("myupcase", ["example value"]))`
- `Deferred` functions can only use types that are built into Puppet (for example `String`). They cannot use types from modules like `stdlib` because Puppet does not plugin-sync these types to the agent.
- Do not use the `Deferred` object as a variable in a string. When compiled, these variables are interpolated, so the *stringified* version of the object would be passed to the agent, instead of the object itself.

Using a template with Deferred values

Templates are rendered on the primary server during catalog compilation. However, this won't work with deferred functions because their values aren't known at compile time. Instead, you need to defer the template rendering.

To defer the template rendering, you need to compile the template source into the catalog, and then render the string using the `inline_epp()` function. The template source must be in the files directory of your module for it to be accessible to the `file()` function.

```
$variables = {
  'password' => Deferred('vault_lookup::lookup',
    ["secret/test", 'https://vault.docker:8200']),
}
```

```
# compile the template source into the catalog
file { ['/etc/secrets.conf':
  ensure => file,
  content => Deferred('inline_epp',
    [file('mymodule/secrets.conf.epp'), $variables]),
}
```

Note: You cannot defer `.erb` style templates like this because of the way they use scope. Use `.epp` templates instead.

Write a deferred function to store secrets

Use the `Deferred` type to create a function that you add to a module to redact sensitive information.

These instructions use Puppet Development Kit (PDK), our recommended tool for creating modules. The steps are also based on RHEL 7 OS.

1. **Install PDK** using the command appropriate to your system.

You might have to restart your command-line interface for `pdk` commands to be in your path.

2. From a working directory, run the following commands. You can accept the default answers to the questions for the steps.
 - a) `pdk new module mymodule`
 - b) `cd mymodule`
 - c) `pdk new class mymodule`
 - d) `mkdir -p lib/puppet/functions`
3. Paste this code into `manifests/init.pp`.

```
# This is a simple example of calling a function at catalog apply time.
#
# @summary Demonstrates calling a Deferred function that is housed with
#           this module in lib/puppet/functions/myupcase.rb
#
# @example
#   puppet apply manifests/init.pp
class mymodule {
  $d = Deferred("mymodule::myupcase", ["mysecret"])

  notify { example :
    message => $d
  }
}

class { 'mymodule': }
```

4. Paste this code into `lib/puppet/functions/mymodule/myupcase.rb`

```
Puppet::Functions.create_function(:'mymodule::myupcase') do
  dispatch :up do
    param 'String', :some_string
  end

  def up(some_string)
    Puppet::Pops::Types::PSensitiveType::Sensitive.new(some_string.upcase)
  end
end
```

5. Run `/opt/puppetlabs/bin/puppet apply manifests/init.pp`. This outputs a notice.

The use of `Sensitive` in the `up` function tells the agent not to store the cleartext value in logs or reports. On the command line and in the Puppet Enterprise console, sensitive data appears as `[redacted]`.

Note: The workflow using `Deferred` functions is the same module adoption workflow that you already use for other modules; you can package functions in a module that are synced down to agents. In most cases, you add the new module to your Puppetfile.

Integrations with secret stores

The Forge already hosts some community modules that provide integrations with secret stores.

Modules with secret store integrations:

- [Azure Key Vault](#): works on the primary server.
- [Cyberark Conjur](#): works on the primary server.
- [Hashicorp Vault](#): works on the agent.
- [Consul Data](#): works on both the agent and the primary server.

Values, data types, and aliases

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

Strings are the most common and useful data type, but you'll also work with others, including numbers, arrays, and some Puppet-specific data types like resource references.

Note that once created, Puppet's values are immutable — they cannot be modified in any way.

For information on type conversion, see [Typecasting](#).

Literal data types as values

Although you'll mostly interact with values *of* the various data types, Puppet also includes values like `String` that *represent* data types.

You can use these special values to examine a piece of data or enforce rules. Usually, they act like patterns, similar to a regular expression: given a value and a data type, you can test whether the value *matches* the data type, and then either adjust your code's behavior accordingly, or raise an error if something has gone wrong.

The pages in this section provide details about using each of the data types as a value. For information about the syntax and behavior of literal data types, see [Data type syntax](#). For information about special abstract data types, which you can use to do more sophisticated or permissive type checking, see [Abstract data types](#).

Puppet's data types

See the following pages to learn more about the syntax, parameters, and usage for each of the data types.

- [Abstract data types](#) on page 508

If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

- [Arrays](#) on page 516

Arrays are ordered lists of values. Resource attributes which accept multiple values (including the relationship metaparameters) generally expect those values in an array. Many functions also take arrays, including the iteration functions.

- [Binary](#) on page 519

A `Binary` object represents a sequence of bytes and it can be created from a `String` in Base64 format, a verbatim `String`, or an `Array` containing byte values. A `Binary` can also be created from a `Hash` containing the value to convert to a `Binary`.

- [Booleans](#) on page 520

Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.

- [Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

- [Default](#) on page 525

Puppet's `default` value acts like a keyword in a few specific usages. Less commonly, it can also be used as a value.

- [Error data type](#) on page 527

An `Error` object contains a non-empty message. It can also contain additional context about why the error occurred.

- [Hashes](#) on page 527

Hashes map keys to values, maintaining the order of the entries according to insertion order.

- [Numbers](#) on page 530

Numbers in the Puppet language are normal integers and floating point numbers.

- [Regular expressions](#) on page 533

A regular expression (sometimes shortened to “regex” or “regexp”) is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

- [Resource and class references](#) on page 535

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

- [Resource types](#) on page 537

Resource types are a special family of data types that behave differently from other data types. They are subtypes of the fairly abstract `Resource` data type. Resource references are a useful subset of this data type family.

- [Sensitive](#) on page 541

Sensitive types in the Puppet language are strings marked as sensitive. The value is displayed in plain text in the catalog and manifest, but is redacted from logs and reports. Because the value is maintained as plain text, use it only as an aid to ensure that sensitive values are not inadvertently disclosed.

- [Strings](#) on page 542

Strings are unstructured text fragments of any length. They're a common and useful data type.

- [Time-related data types](#) on page 550

A `Timespan` defines the length of a duration of time, and a `Timestamp` defines a point in time. For example, “two hours” is a duration that can be represented as a `Timespan`, while “three o'clock in the afternoon UTC on 8 November, 2018” is a point in time that can be represented as a `Timestamp`. Both types can use nanosecond values if it is available on the platform.

- [Undef](#) on page 552

Puppet's `undef` value is roughly equivalent to `nil` in Ruby. It represents the absence of a value. If the `strict_variables` setting isn't enabled, variables which have never been declared have a value of `undef`.

Type aliases

Type aliases allow you to create reusable and descriptive data types and resource types.

By using type aliases, you can:

- Give a type a descriptive name, such as `IPv6Addr`, instead of creating or using a complex pattern-based type.
- Shorten and move complex type expressions.
- Improve code quality by reusing existing types instead of inventing new types.
- Test type definitions separately from manifests.

Type aliases are transparent, which means they are fully equivalent to the types of which they are aliases. For example, in the following code, the `notice` returns `true` because `MyType` is an alias of the `Integer` type:

```
type MyModule::MyType = Integer
notice MyModule::MyType == Integer
```

Note: The internal types `TypeReference` and `TypeAlias` are never values in Puppet code .

Creating type aliases

Use the following syntax to create a type alias:

```
type <MODULE NAME>::<ALIAS NAME> = <TYPE DEFINITION>
```

The `<MODULE NAME>` must be named after the module that contains the type alias, and both the `<MODULE NAME>` and `<ALIAS NAME>` begin with a capital letter and must not be a [reserved word](#).

For example, you can create a type alias named `MyType` that is equivalent to the `Integer` data type:

```
type MyModule::MyType = Integer
```

You can then declare a parameter using the alias as though it were a unique data type:

```
MyModule::MyType $example = 10
```

To make your code easier to maintain and troubleshoot, store type aliases as `.pp` files in your module's `types` directory, which is a top-level directory and sibling of the `manifests` and `lib` directories. Define only one alias per file, and name the file after the type alias name converted to lowercase. For example, `MyType` is expected to be loaded from a file named `mytype.pp`.

You can create recursive type aliases, which can refer to the alias being declared or to other types, thereby defining complex, descriptive type definitions without using the `Any` type. For example:

```
type MyModule::Tree = Array[Variant[Data, Tree]]
```

This `Tree` type alias is defined as a being built out of Arrays that contain `Data`, or a `Tree`:

```
[1,2 [3], [4, [5, 6], [[[[1,2,3]]]]]]
```

You can also create aliases to resource types:

```
type MyModule::MyFile = File
```

When defining an alias to a resource type, use its short form (for example, `File`) instead of its long form (`Resource[File]`).

Related information

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Resources](#) on page 383

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

Typecasting

Typecasting is a method for converting data from one data type to another. In many cases, you can explicitly typecast into the type you need, but the Puppet language also provides shortcuts for common data conversions, as well as support for more complex conversions.

Creating a new typed variable

When there's an unambiguous translation between one data type and another, you can typecast into the type you need:

```
$temp    = Float("98.6")      # Cast into a float variable; 98.6
$count   = Integer("42")     # Cast into an integer variable; 42
$bool    = Boolean("false")  # Cast into a boolean false
$string  = String(100)       # Cast into the string "100"
```

You can also typecast between number formats, such as turning a hex number string into an integer:

```
$intval = Integer("0xFF")    # Cast into an integer variable; 255
```

Invoking the data type directly is a shortcut for calling the `.new()` function. For more information, including the conversion rules and options available, see [the function reference for new\(\)](#).

Automatic coercions

When you use strings in arithmetic, Puppet assumes that you did so intentionally and coerces them to the proper types. For example:

```
$result = "2" + "2"          # Cast into an integer variable; 4
$multiple = 4 * "25"        # Cast into an integer variable; 100
$float    = "2.75" * 2      # Cast into a float variable; 5.5
$identity = "1024" + 0      # Cast into integer variable; 1024 (direct
                             conversion)
```

When you interpolate variables into a string, Puppet converts them to their string representation using the most obvious method. If you need an explicit conversion, you can manually typecast. For example:

```
$var = 1
notice("The truth value of ${var} is ${Boolean($var)}") # Outputs "The truth
value of 1 is true"
```

Extracting a Number from a String fragment

To convert a string that contains non-numeric characters into a number, for example `32°` or `9,000`, you must parse the string. You can parse a string using [the `scanf` function](#).

For example:

```
$input  = "It is 32° outside!"
$format = "It is %i° outside!"
scanf($input, $format) # returns array of matches; [32]

$input  = "The melting point of iron is 2,800°F".delete(",")
$format = "The melting point of iron is %i°F"
scanf($input, $format) # returns array of matches; [2800]
```

Here are more example format conversions:

Conversion	Description of match
%d, %u	Optionally signed decimal integer
%i	Optionally signed integer with auto-detected base
%o	Optionally signed octal integer
%x, %X	Optionally signed hexadecimal integer
%s	String, or a sequence of non-white-space characters
%c	A single character
%%	A % character

Converting to a Boolean

You can cast case-insensitive strings of `true/false`, `yes/no`, *non-zero*/0, and integers of *non-zero*/0:

```
$value = Boolean('true')      # true
$value = Boolean('Yes')       # true
$value = Boolean('FALSE')     # false
$value = Boolean('1')         # true
$value = Boolean(0)           # false
$value = Boolean(127)         # true
```

Converting data structures

Converting a hash to an array creates a multi-dimensional array by flattening each key-value pair into a two-element array as an element of another array. Converting an array to a hash interleaves alternating elements as keys and values.

```
$an_array = Array({a => 10, b => 20}) # results in [[a, 10],[b, 20]]
$a_hash   = Hash([1,2,3,4])          # results in {1=>2, 3=>4}
$a_hash   = Hash([[1,2],[3,4]])      # results in {1=>2, 3=>4}
```

The Array conversion also takes a second argument to "force" an array conversion. This slightly changes the semantics of the operation by creating a single-element array out of the first argument if it's not already.

```
$an_array = Array(1, true)           # results in [1]
$an_array = Array([1], true)         # results in [1]
$an_array = Array(1)                 # Raises a type error, cannot convert
directly
$an_array = Array({1 => 2}, true)     # results in [{1 => 2}]
$an_array = Array({1 => 2})           # results in [[1, 2]]
```

Abstract data types

If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Each of Puppet's core data types has a corresponding value that represents that data type, which can be used to match values of that type in several contexts. Each of those core data types only match a particular set of values. They let you further restrict the values they'll match, but only in limited ways, and there's no way to *expand* the set of values they'll match. If you need to do this, use the corresponding abstract data type.

Related information

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Values, data types, and aliases](#) on page 504

Most of the things you can do with the Puppet language involve some form of data. An individual piece of data is called a *value*, and every value has a *data type*, which determines what kind of information that value can contain and how you can interact with it.

Flexible data types

These abstract data types can match values with a variety of concrete data types. Some of them are similar to a concrete type but offer alternate ways to restrict them (for example, `Enum`), and some of them let you combine types and match a union of what they would individually match (for example, `Variant` and `Optional`).

The Optional data type

The `Optional` data type wraps *one* other data type, and results in a data type that matches anything that type would match *plus* `undef`. This is useful for matching values that are allowed to be absent. It takes one required parameter.

The full signature for `Optional` is:

```
Optional[<DATA TYPE>]
```

Position	Parameter	Data type	Default value	Description
1	Data type	Type or String	none (you must specify a value)	The data type to add <code>undef</code> to.

If you specify a string `"my_string"` as the parameter, it's equivalent to using `Optional[Enum["my_string"]]` — it matches only that exact string value or `undef`.

`Optional[<DATA TYPE>]` is equivalent to `Variant[<DATA TYPE>, Undef]`.

Examples:

`Optional[String]`

Matches any string or `undef`.

`Optional[Array[Integer[0, 10]]]`

Matches an array of integers between 0 and 10, or `undef`.

`Optional["present"]`

Matches the exact string `"present"` or `undef`.

The NotUndef data type

The `NotUndef` type matches any value except `undef`. It can also wrap one other data type, resulting in a type that matches anything the original type would match except `undef`. It accepts one optional parameter.

The full signature for `NotUndef` is:

```
NotUndef[<DATA TYPE>]
```

Position	Parameter	Data type	Default value	Description
1	Data type	Type or String	Any	The data type to subtract <code>undef</code> from.

If you specify a string as a parameter for `NotUndef`, it's equivalent to writing `NotUndef[Enum["my string"]]` — it matches only that exact string value. This doesn't actually subtract anything, because the `Enum` wouldn't have matched `undef` anyway, but it's a convenient notation for mandatory keys in `Struct` schema hashes.

The Variant data type

The `Variant` data type combines any number of other data types, and results in a type that matches the union of what any of those data types would match. It takes any number of parameters, and requires at least one.

The full signature for `Variant` is:

```
Variant[ <DATA TYPE>, (<DATA TYPE>, ...) ]
```

Position	Parameter	Data type	Default value	Description
1 and up	Data type	Type	none (required)	A data type to add to the resulting compound data type. You must provide at least one data type parameter, and can provide any number of additional ones.

Examples:

`Variant[Integer, Float]`

Matches any integer or floating point number (equivalent to `Numeric`).

`Variant[Enum['true', 'false'], Boolean]`

matches `'true'`, `'false'`, `true`, or `false`.

The Pattern data type

The `Pattern` data type only matches strings, but it provides an alternate way to restrict which strings it matches. It takes any number of regular expressions, and results in a data type that matches any strings that would match any of those regular expressions. It takes any number of parameters, and requires at least one.

The full signature for `Pattern` is:

```
Pattern[ <REGULAR EXPRESSION>, (<REGULAR EXPRESSION>, ...) ]
```

Position	Parameter	Data type	Default value	Description
1 and up	Regular expression	Regex	none (required)	A regular expression describing a set of strings that the resulting data type should match. You must provide at least one regular expression parameter, and can provide any number of additional ones.

You can use capture groups in the regular expressions, but they won't cause any variables, like `$1`, to be set.

Examples:

Pattern[/\A[a-z].*/]

Matches any string that begins with a lowercase letter.

Pattern[/\A[a-z].*/ , /\ANone\Z/]

Matches the above or the exact string None.

The Enum data type

The Enum data type only matches strings, but it provides an alternate way to restrict which strings it matches. It takes any number of strings, and results in a data type that matches any string values that exactly match one of those strings. Unlike the == operator, this matching is case-sensitive. It takes any number of parameters, and requires at least one.

The full signature for Enum is:

```
Enum[ <OPTION>, ( <OPTION>, ... ) ]
```

Position	Parameter	Data type	Default value	Description
1 and up	Option	String	none (required)	One of the literal string values that the resulting data type should match. You must provide at least one option parameter, and can provide any number of additional ones.

Examples:

Enum['stopped', 'running']

Matches the strings 'stopped' and 'running', and no other values.

Enum['true', 'false']

Matches the strings 'true' and 'false', and no other values. Does not match the boolean values true or false (without quotes).

The Tuple data type

The Tuple type only matches arrays, but it lets you specify different data types for every element of the array, in order. It takes any number of parameters, and requires at least one.

The full signature for Tuple is:

```
Tuple[ <CONTENT TYPE>, ( <CONTENT TYPE>, ..., <MIN SIZE>, <MAX SIZE> ) ]
```

Position	Parameter	Data type	Default value	Description
1 and up	Content type	Type	none (required)	What kind of values the array contains at the given position. You must provide at least one content type parameter, and can provide any number of additional ones.

Position	Parameter	Data type	Default value	Description
-2 (second-last)	Minimum size	Integer	number of content types	The minimum number of elements in the array. If this is smaller than the number of content types you provided, any elements beyond the minimum are optional; however, if present, they must still match the provided content types. This parameter accepts the value <code>default</code> , but this won't use the default value; instead, it means 0 (all elements optional).
-1 (last)	Maximum size	Integer	number of content types	<p>The maximum number of elements in the array. You cannot specify a maximum without also specifying a minimum. If the maximum is larger than the number of content types you provided, it means the array can contain any number of additional elements, which all must match the last content type. This parameter accepts the value <code>default</code>, but this won't use the default value; instead, it means infinity (any number of elements matching the final content type).</p> <p>Don't set the maximum smaller than the number of content types you provide.</p>

Examples:

`Tuple[String, Integer]`

Matches a two-element array containing a string followed by an integer, like ["hi" , 2].

`Tuple[String, Integer, 1]`

Matches the above **or** a one-element array containing only a string.

`Tuple[String, Integer, 1, 4]`

Matches an array containing one string followed by zero to three integers.

`Tuple[String, Integer, 1, default]`

Matches an array containing one string followed by any number of integers.

The Struct data type

The `Struct` type only matches hashes, but it lets you specify:

- The name of every allowed key.
- Whether each key is required or optional.
- The allowed data type for each of those keys' values.

It takes one mandatory parameter.

The full signature for `Struct` is:

```
Struct[<SCHEMA HASH>]
```

Position	Parameter	Data type	Default value	Description
1	Schema hash	Hash[Variant[String (required), Optional, NotUndef], Type]		A hash that has all of the allowed keys and data types for the struct.

A `Struct`'s schema hash must have the same keys as the hashes it matches. Each value must be a data type that matches the allowed values for that key.

The keys in a schema hash are usually strings. They can also be an `Optional` or `NotUndef` type with the key's name as their parameter.

If a key is a string, Puppet uses the *value*'s type to determine whether it's optional — because accessing a missing key resolves to the value `undef`, the key is optional if the value type accepts `undef` (like `Optional[Array]`).

Note that this doesn't distinguish between an explicit value of `undef` and an absent key. If you want to be more explicit, you can use `Optional['my_key']` to indicate that a key can be absent, and `NotUndef['my_key']` to make it mandatory. If you use one of these, a value type that accepts `undef` is only used to decide about explicit `undef` values, not missing keys.

The following example `Struct` matches hashes like {mode => 'read', path => '/etc/fstab'}. Both the mode and path keys are mandatory; mode's value must be one of 'read', 'write', or 'update', and path must be a string of at least one character:

```
Struct[{mode => Enum[read, write, update],
       path => String[1]}]
```

The following data type would match the same values as the previous example, but the path key is optional. If present, path must match `String[1]` or `Undef`:

```
Struct[{mode => Enum[read, write, update],
       path => Optional[String[1]]}]
```

In the following data type, the `owner` key can be absent, but if it's present, it must be a string; a value of `undef` isn't allowed:

```
Struct[{mode          => Enum[read, write, update],
        path          => Optional[String[1]],
        Optional[owner] => String[1]}]
```

In the following data type, the `owner` key is mandatory, but it allows an explicit `undef` value:

```
Struct[{mode          => Enum[read, write, update],
        path          => Optional[String[1]],
        NotUndef[owner] => Optional[String[1]]}]
```

Parent types

These abstract data types are the parents of multiple other types, and match values that would match *any* of their subtypes. They're useful when you have very loose restrictions but not *no* restrictions.

The `Scalar` data type

The `Scalar` data type matches *all* values of the following concrete data types:

- Numbers (both integers and floats)
- Strings
- Booleans
- Regular expressions

It doesn't match `undef`, `default`, resource references, arrays, or hashes. It takes no parameters.

`Scalar` is equivalent to `Variant[Integer, Float, String, Boolean, Regexp]`.

The `ScalarData` data type

The `ScalarData` data type represents a restricted set of "value" data types that have concrete direct representation in JSON.

`ScalarData` is an alias for `Variant[Integer, Float, String, Boolean]`.

The `Data` data type

The `Data` data type matches any value that would match `Scalar`, but it also matches:

- `undef`
- Arrays that only contain values that also match `Data`
- Hashes whose keys match `Scalar` and whose values also match `Data`

It doesn't match `default` or resource references. It takes no parameters.

`Data` is especially useful because it represents the subset of types that can be directly represented in almost all serialization format, such as JSON.

The `Collection` data type

The `Collection` type matches *any* array or hash, regardless of what kinds of values or keys it contains. It only partially overlaps with `Data`—there are values, such as an array of resource references, that match `Collection` but do not match `Data`.

`Collection` is equivalent to `Variant[Array[Any], Hash[Any, Any]]`.

The `CatalogEntry` data type

The `CatalogEntry` data type is the parent type of `Resource` and `Class`. Like those types, the Puppet language contains no values that it ever matches. However, the type `Type[CatalogEntry]` matches any class reference or resource reference. It takes no parameters.

The `Any` data type

The `Any` data type matches *any* value of *any* data type.

The `Iterable` data type

The `Iterable` data type represents all data types that can be iterated; in other words, all data types where the value is some kind of container of individual values. The `Iterable` type is abstract in that it does not specify if it represents a concrete data type (such as `Array`) that has storage in memory, or if it is an algorithmic construct like a transformation function (such as the `step` function).

The `Iterator` data type

The `Iterator` data type is an `Iterable` that does not have a concrete backing data type holding a copy of the values it will produce when iterated over. It represents an algorithmic transformation of some source (which in turn can be algorithmic).

An `Iterator` may not be assigned to an attribute of a resource, and it may not be used as an argument to version 3.x functions. To create a concrete value an `Iterator` must be "rolled out" by using a function at the end of a chain that produces a concrete value.

The `RichData` data type

The `RichData` data type represents the abstract notion of "serializeable" and includes all the types in the type system except `Runtime`, `Callable`, `Iterator`, and `Iterable`. It is expressed as an alias of `Variant[Default, Object, Scalar, SemVerRange, Type, Undef, Array[RichData], Hash[RichData, RichData]]`.

Other types

These types aren't quite like the others.

The `Callable` data type

The `Callable` data type matches callable lambdas provided as function arguments.

There is no way to interact with `Callable` values in the Puppet language, but Ruby functions written to the function API (`Puppet::Functions`) can use `Callable` to inspect the lambda provided to the function.

The full signature for `Callable` is:

```
Callable[ (<DATA TYPE>, ...,) <MIN COUNT>, <MAX COUNT>, <BLOCK TYPE> ]
```

All of these parameters are optional.

Position	Parameter	Data type	Default value	Description
1 to n	Data type	Type	none	Any number of data types, representing the data type of each argument the lambda accepts.

Position	Parameter	Data type	Default value	Description
-3 (third last)	Minimum count	Integer	0	The minimum number of arguments the lambda accepts. This parameter accepts the value <code>default</code> , which uses its default value 0.
-2 (second last)	Maximum count	Integer	infinity	The maximum number of arguments the lambda accepts. This parameter accepts the value <code>default</code> , which uses its default value, infinity.
-1 (last)	Block type	Type[Callable]	none	The <code>block_type</code> of the lambda.

Arrays

Arrays are ordered lists of values. Resource attributes which accept multiple values (including the relationship metaparameters) generally expect those values in an array. Many functions also take arrays, including the iteration functions.

Arrays are written as comma-separated lists of values surrounded by square brackets, `[]`. An optional trailing comma is allowed between the final value and the closing square bracket.

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

The values in an array can be any data type.

Array values are immutable — they cannot be changed once they are created.

Accessing items in an array

You can access items in an array by their numerical index, counting from zero. Square brackets are used for accessing. For example:

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

This manifest would log `two` as a notice. The value of `$my_array[0]` would be `one`, because indexes count from zero.

The opening square bracket must not be preceded by white space:

```
$my_array = [ 'one', 'two', 'three', 'four', 'five' ]
notice( $my_array[2] ) # ok
notice( $my_array [2] ) # syntax error
```

Nested arrays and hashes can be accessed by chaining indexes:

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

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

Arrays support negative indexes, counting backward from the last element, with `-1` being the last element of the array:

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

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

If you try to access an element beyond the bounds of the array, its value is `undef`:

```
$my_array = [ 'one', 'two', 'three', 'four', 'five' ]
$cool_value = $my_array[6] # value is undef
```

When testing with a regular expression whether an `Array[<TYPE>]` data type matches a given array, empty arrays match if the type accepts zero-length arrays.

```
$my_array = []
if $my_array =~ Array[String] {
  notice( 'my_array' )
}
```

This manifest would log `my_array` as a notice, because the expression matches the empty array.

Accessing sections of an array

You can access sections of an array by numerical index. Like accessing individual items, accessing sections uses square brackets, but it uses two indexes, separated by a comma. For example: `$array[3,10]`.

The result of an array section is always another array.

The first number of the index is the start position. Positive numbers count forward from the start of the array, starting at zero. Negative numbers count backward from the end of the array, starting at `-1`.

The second number of the index is the stop position. Positive numbers are lengths, counting forward from the start position. Negative numbers are absolute positions, counting backward from the end of the array starting at `-1`.

For example:

```
$my_array = [ 'one', 'two', 'three', 'four', 'five' ]
notice( $my_array[2,1] ) # evaluates to ['three']
notice( $my_array[2,2] ) # evaluates to ['three', 'four']
notice( $my_array[2,-1] ) # evaluates to ['three', 'four', 'five']
notice( $my_array[-2,1] ) # evaluates to ['four']
```

Array operators

You can use operators with arrays to create new arrays:

- append with `<<`
- concatenate with `+`
- remove elements with `-`

or to convert arrays to comma-separated lists with `*` (splat). See the Array operators section of [Expressions](#) for more information.

Additional functions for arrays

The `puppetlabs-stdlib` module contains useful functions for working with arrays, including:

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

Related information

[Regular expressions](#) on page 533

A regular expression (sometimes shortened to “regex” or “regexp”) is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

[Undef](#) on page 552

Puppet's undef value is roughly equivalent to nil in Ruby. It represents the absence of a value. If the `strict_variables` setting isn't enabled, variables which have never been declared have a value of undef.

The Array data type

The data type of arrays is `Array`. By default, `Array` matches arrays of any length, provided all values in the array match the abstract data type `Data`. You can use parameters to restrict which values `Array` matches.

Parameters

The full signature for `Array` is:

```
Array[<CONTENT TYPE>, <MIN SIZE>, <MAX SIZE>]
```

These parameters are optional. They must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

Position	Parameter	Data type	Default value	Description
1	Content type	Type	Data	The kind of values the array contains. You can specify only one data type per array, and every value in the array must match that type. Use an abstract type to allow multiple data types. If the order of elements matters, use the <code>Tuple</code> type instead of <code>Array</code> .
2	Minimum size	Integer	0	The minimum number of elements in the array. This parameter accepts the special value <code>default</code> , which uses its default value.

Position	Parameter	Data type	Default value	Description
3	Maximum size	Integer	infinite	The maximum number of elements in the array. This parameter accepts the special value <code>default</code> , which uses its default value.

Examples:

Array

Matches an array of any length. All elements in the array must match `Data`.

Array[String]

Matches an array of any size that contains only strings.

Array[Integer, 6]

Matches an array containing at least six integers.

Array[Float, 6, 12]

Matches an array containing at least six and at most 12 floating-point numbers.

Array[Variant[String, Integer]]

Matches an array of any size that contains only strings or integers, or both.

Array[Any, 2]

Matches an array containing at least two elements, allowing any data type (including `Type` and `Resource`).

The abstract `Tuple` data type lets you specify data types for every element in an array, in order. Abstract types, such as `Variant` and `Enum`, are useful when specifying content types for arrays that include multiple kinds of data.

Related information

[Abstract data types](#) on page 508

If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Binary

A `Binary` object represents a sequence of bytes and it can be created from a `String` in Base64 format, a verbatim `String`, or an `Array` containing byte values. A `Binary` can also be created from a `Hash` containing the value to convert to a `Binary`.

Binary algebra

- A `Binary` can only be assigned a `Binary` value
- A `Binary` has no type parameters

Binary.new

The signatures are:

```
type ByteInteger = Integer[0,255]
type Base64Format = Enum["%b", "%u", "%B", "%s"]
type StringHash = Struct[{value => String, "format" =>
  Optional[Base64Format]}]
type ArrayHash = Struct[{value => Array[ByteInteger]}]
type BinaryArgsHash = Variant[StringHash, ArrayHash]
```

```
function Binary.new(
  String $base64_str,
  Optional[Base64Format] $format
)

function Binary.new(
  Array[ByteInteger] $byte_array
}

# Same as for String, or for Array, but where arguments are given in a
Hash.
function Binary.new(BinaryArgsHash $hash_args)
```

The format codes represent the following data encoding formats:

Format	Explanation
%b	The data is in base64 encoding. Padding required by base64 strict encoding is added by default.
%u	The data is in URL-safe base64 encoding.
%B	The data is in base64 strict encoding.
%s	The data is a Puppet string. If the string is not valid UTF-8 or convertible to UTF-8, Puppet raises an error.
%r	The data is raw Ruby. The byte sequence in the given string is used verbatim irrespective of possible encoding errors.

- The default format code is %B.
- Avoid using the raw Ruby format %r. It exists for backward compatibility when a string received from some function should be treated as Binary. Such code should be changed to return a Binary instead of a String. This format will be deprecated in a future version of the specification when enough time has been given to migrate existing use of Ruby "binary strings", or Ruby strings that do not report accurate encoding.

Example: Creating Binary-typed content

```
# create the binary content "abc" from base64 encoded string
$a = Binary('YWJj')

# create the binary content from content in a module's file
$b = binary_file('mymodule/mypicture.jpg')
```

Booleans

Booleans are one-bit values, representing true or false. The condition of an `if` statement expects an expression that resolves to a boolean value. All of Puppet's comparison operators resolve to boolean values, as do many functions.

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

Automatic conversion to boolean

If a non-boolean value is used where a boolean is required:

- The `undef` value is converted to boolean `false`.
- All other values are converted to boolean `true`.

Notably, this means the string values `" "` (a zero-length string) and `"false"` (in quotation marks) both resolve to `true`.

To convert values to booleans with more permissive rules (for example, 0 to false, or "false" to false), use the `str2bool` and `num2bool` functions in the `puppetlabs-stdlib` module.

The Boolean data type

The data type of boolean values is `Boolean`.

It matches only the values `true` or `false`, and accepts no parameters.

You can use abstract types to match values that might be boolean or might have some other value. For example, `Optional[Boolean]` matches `true`, `false`, or `undef`. `Variant[Boolean, Enum["true", "false"]]` matches stringified booleans as well as true booleans.

Related information

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Abstract data types](#) on page 508

If you’re using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Data type syntax

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

You can use these special values to examine a piece of data or enforce rules. For example, you can test whether something is a string with the expression `$possible_string =~ String`, or specify that a class parameter requires string values with `class myclass (String $string_parameter = "default value") { ... }`.

Syntax

Data types are written as unquoted upper-case words, like `String`.

Data types sometimes take parameters, which make them more specific. For example, `String[8]` is the data type of strings with a minimum of eight characters.

Each known data type defines how many parameters it accepts, what values those parameters take, and the order in which they must be given. Some of the abstract types *require* parameters, and most types have some optional parameters available.

The general form of a data type is:

- An upper-case word matching one of the known data types.
- Sometimes, a set of parameters, which consists of:
 - An opening square bracket `[` after the type’s name. There can’t be any space between the name and the bracket.
 - A comma-separated list of values or expressions. Arbitrary whitespace is allowed, but you can’t have a trailing comma after the final value.
 - A closing square bracket `]`.

The following example uses an abstract data type `Variant`, which takes any number of data types as parameters. One of the parameters provided in the example is another abstract data type `Enum`, which takes any number of strings as parameters:

```
Variant[Boolean, Enum['true', 'false', 'running', 'stopped']]
```

Note: When parameters are required, you must specify them. The only situation when you can leave out required parameters is if you're referring to the type itself. For example, `Type[Variant]` is legal, even though `Variant` has required parameters.

Usage

Data types are useful in parameter lists, match (`=~`) expressions, case statements, and selector expressions. There are also a few less common uses for them.

Specify data types in your Puppet code whenever you can, aligning them in columns. Type your class parameters wherever possible, and be specific when using a type. For example, use an `Enum` for input validation, instead of using a `String` and checking the contents of the string in the code. You have the option to specify `String[1]` instead of `String`, because you might want to enforce non-empty strings. Specify data types as deeply as possible, without affecting readability. If readability becomes a problem, consider [creating a custom data type alias](#).

Parameter lists

Classes, defined types, and lambdas all let you specify parameters, which let your code request data from a user or some other source. Generally, your code expects each parameter to be a specific kind of data. You can enforce that expectation by putting a data type before that parameter's name in the parameter list. At evaluation time, Puppet raises an error if a parameter receives an illegal value.

For example, consider the following class. If you tried to set `$autoupdate` to a string like `"true"`, Puppet would raise an error, because it expects a `Boolean` value:

```
class ntp (
  Boolean $service_manage = true,
  Boolean $autoupdate      = false,
  String  $package_ensure = 'present',
  # ...
) {
  # ...
}
```

Abstract data types let you write more sophisticated and flexible restrictions. For example, this `$puppetdb_service_status` parameter accepts values of `true`, `false`, `"true"`, `"false"`, `"running"`, and `"stopped"`, and raises an error for any other value:

```
class puppetdb::server (
  Variant[Boolean, Enum['true', 'false', 'running', 'stopped']]
  $puppetdb_service_status = $puppetdb::params::puppetdb_service_status,
) inherits puppetdb::params {
  # ...
}
```

Cases

Case statements and selector expressions allow data types as their cases. Puppet chooses a data type case if the control expression resolves to a value of that data type. For example:

```
$enable_real = $enable ? {
  Boolean => $enable,
  String  => str2bool($enable),
  Numeric => num2bool($enable),
  default => fail('Illegal value for $enable parameter'),
}
```

Match expressions

The match operators `=~` and `!~` accept a data type on the right operand, and test whether the left operand is a value of that data type.

For example, `5 =~ Integer` and `5 =~ Integer[1,10]` both resolve to `true`.

Less common uses

The built-in function `assert_type` takes a value and a data type, and raises errors if your code encounters an illegal value. Think of it as shorthand for an `if` statement with a non-match (`!~`) expression and a `fail()` function call.

You can also provide data types as both operands for the comparison operators `==`, `!=`, `<`, `>`, `<=`, and `>=`, to test whether two data types are equal, whether one is a subset of another, and so on.

Obtaining data types

The built-in function `type` returns the type of any value. For example, `type(3)` returns `Integer[3,3]`.

List of Puppet data types

The following data types are available in the Puppet language.

For details on each data type, see the linked documentation or the [specification document](#).

Data type	Purpose	Type category
Any	The parent type of all types.	Abstract
Array	The data type of arrays.	Data
Binary	A type representing a sequence of bytes.	Data
Boolean	The data type of Boolean values.	Data, Scalar
Callable	Something that can be called (such as a function or lambda).	Platform
CatalogEntry	The parent type of all types that are included in a Puppet catalog.	Abstract
Class	A special data type used to declare classes.	Catalog
Collection	A parent type of <code>Array</code> and <code>Hash</code> .	Abstract
Data	A parent type of all data directly representable as JSON.	Abstract
Default	The "default value" type.	Platform
Deferred	A type describing a call to be resolved in the future.	Platform
Enum	An enumeration of strings.	Abstract
Error	A type used to communicate when a function has produced an error.	
Float	The data type of floating point numbers.	Data, Scalar
Hash	The data type of hashes.	Data
Init	A type used to accept values that are compatible of some other type's "new".	
Integer	The data type of integers.	Data, Scalar

Data type	Purpose	Type category
Iterable	A type that represents all types that allow iteration.	Abstract
Iterator	A special kind of lazy Iterable suitable for chaining.	Abstract
NotUndef	A type that represents all types not assignable from the Undef type.	Abstract
Numeric	The parent type of all numeric data types.	Abstract
Object	Experimental. Can be a simple object only having attributes, or a more complex object also supporting callable methods.	
Optional	Either Undef or a specific type.	Abstract
Pattern	An enumeration of regular expression patterns.	Abstract
Regexp	The data type of regular expressions.	Scalar
Resource	A special data type used to declare resources.	Catalog
RichData	A parent type of all data types except the non serializeable types Callable, Iterator, Iterable, and Runtime.	Abstract
Runtime	The type of runtime (non Puppet) types.	Platform
Scalar	Represents the abstract notion of "value".	Abstract
ScalarData	A parent type of all single valued data types that are directly representable in JSON.	Abstract
SemVer	A type representing semantic versions.	Scalar
SemVerRange	A range of SemVer versions.	Abstract
Sensitive	A type that represents a data type that has "clear text" restrictions.	Platform
String	The data type of strings.	Data, Scalar
Struct	A Hash where each entry is individually named and typed.	Abstract
Timespan	A type representing a duration of time.	Scalar
Timestamp	A type representing a specific point in time	Scalar
Tuple	An Array where each slot is typed individually	Abstract

Data type	Purpose	Type category
Type	The type of types.	Platform
Typeset	Experimental. Represents a collection of Object-based data types.	
Undef	The "no value" type.	Data, Platform
URI	A type representing a Uniform Resource Identifier	Data
Variant	One of a selection of types.	Abstract

The `Type` data type

The data type of literal data type values is `Type`. By default, `Type` matches any value that represents a data type, such as `Integer`, `Integer[0,800]`, `String`, or `Enum["running", "stopped"]`. You can use parameters to restrict which values `Type` matches.

Parameters

The full signature for `Type` is:

```
Type[ <ANY DATA TYPE> ]
```

This parameter is optional.

Position	Parameter	Data type	Default	Description
1	Any data type	<code>Type</code>	Any	A data type, which causes the resulting <code>Type</code> object to only match against that type or types that are more specific subtypes of that type.

Examples:

`Type`

Matches any data type, such as `Integer`, `String`, `Any`, or `Type`.

`Type[String]`

Matches the data type `String`, as well as any of its more specific subtypes like `String[3]` or `Enum["running", "stopped"]`.

`Type[Resource]`

Matches any `Resource` data type — that is, any resource reference.

Default

Puppet's `default` value acts like a keyword in a few specific usages. Less commonly, it can also be used as a value.

Syntax

The only value in the default data type is the bare word `default`.

Usage with cases and selectors

In [case statements and selector expressions](#), you can use `default` as a case. Puppet attempts to match a `default` case last, after it has tried to match against every other case.

Usage with per-block resource defaults

You can use `default` as the title in a [resource declaration](#) to invoke a particular behavior. Instead of creating a resource and adding it to the catalog, the `default` resource sets fallback attributes that can be used by any other resource in the same resource expression.

In the following example, all of the resources in the block inherits attributes from `default` unless they specifically override them:

```
file {
  default:
    ensure => file,
    mode   => '0600',
    owner  => 'root',
    group  => 'root',
  ;
  '/etc/ssh_host_dsa_key':
  ;
  '/etc/ssh_host_key':
  ;
  '/etc/ssh_host_dsa_key.pub':
    mode => '0644',
  ;
  '/etc/ssh_host_key.pub':
    mode => '0644',
  ;
}
```

For more information, see [Resources \(Advanced\)](#).

Usage as parameters of data types

Several data types take parameters that have default values. In some cases, like minimum and maximum sizes, the default value can be difficult or impossible to refer to using the available literal values in the Puppet language. For example, the default value of the `String` type's maximum length parameter is infinity, which can't be represented in the Puppet language.

These parameters let you provide a value of `default` to indicate that you want the default value.

Other default usage

You can use the value `default` anywhere you aren't prohibited from using it. In these cases, it generally won't have any special meaning.

There are a few reasons you might want to do this. A prime example is if you are writing a class or defined resource type and want to give users the option to specifically request a parameter's default value. Some people have used `undef` to do this, but that's no good when dealing with parameters where `undef` would, itself, be a meaningful value. Others have used a value like the string `"UNSET"`, but this can be messy.

Using `default` in this scenario lets you distinguish among:

- A chosen “real” value.
- A chosen value of `undef`.
- Explicitly declining to choose a value, represented by `default`.

In other other words, `default` can be useful when you need a truly meaningless value.

The `Default` data type

The data type of `default` is `Default`. It matches only the value `default`, and takes no parameters.

Example:

Variant[String, Default, Undef]

Matches undef, default, or any string.

Related information

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Abstract data types](#) on page 508

If you’re using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Error data type

An `Error` object contains a non-empty message. It can also contain additional context about why the error occurred.

Parameters

The full signature for `Error` is:

```
Error[<MESSAGE>, <KIND>, <DETAILS>, <ISSUE CODE>]
```

The following table shows the available parameters. Note that the message parameter is required and the others are optional.

Position	Parameter	Data type	Default value	Description
1	Message	<code>String[1]</code>		The error message of length greater than 0.
2	Kind	<code>Optional[String[1]]</code>	<code>undef</code>	The kind or category of error.
3	Details	<code>Optional[Hash[String[1], undef]]</code>	<code>undef</code>	Additional details about the error.
4	Issue code	<code>Optional[String[1]]</code>	<code>undef</code>	A unique code identifying the issue which can be used to translate error messages into different locales.

Accessing values

The `Error` object implements methods for retrieving additional information. For example:

```
notice($err.msg)
notice($err.kind)
notice($err.issue_code)
notice($err.details)
```

Hashes

Hashes map keys to values, maintaining the order of the entries according to insertion order.

Syntax

Hashes are written as a pair of curly braces `{ }` containing any number of key-value pairs. A key is separated from its value by an arrow (sometimes called a fat comma or hash rocket) `=>`, and adjacent pairs are separated by commas. An optional trailing comma is allowed between the final value and the closing curly brace.

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

```
{ key1 => 'val1', key2 => 'val2', }
```

Hash keys can be any data type, but generally, you should use only strings. Put quotation marks around keys that are strings. Don't assign a hash with non-string keys to a resource attribute or class parameter, because Puppet cannot serialize non-string hash keys into the catalog.

```
{ 'key1' => ['val1','val2'],
  'key2' => { 'key3' => 'val3', },
  'key4' => true,
  'key5' => 12345,
}
```

Accessing values

Access hash members with their key inside square brackets:

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

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

If you try to access a nonexistent key from a hash, its value is `undef`:

```
$cool_value = $myhash[absent_key] # Value is undef
```

Nested arrays and hashes can be accessed by chaining indexes:

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

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

Merging hashes

When hashes are merged (using the addition (+) operator), the keys in the constructed hash have the same order as in the original hashes, with the left hash keys ordered first, followed by any keys that appeared only in the hash on the right side of the merge.

Where a key exists in both hashes, the merged hash uses the value of the key in the hash to the right of the addition operator (+). For example:

```
$values = {'a' => 'a', 'b' => 'b'}
$overrides = {'a' => 'overridden'}
$result = $values + $overrides
notice($result)
-> {'a' => 'overridden', 'b' => 'b'}
```


The Hash data type

The data type of hashes is `Hash`. By default, `Hash` matches hashes of any size, as long as their keys match the abstract type `Scalar` and their values match the abstract type `Data`. You can use parameters to restrict which values `Hash` matches.

Parameters

The full signature for `Hash` is:

```
Hash[ <KEY TYPE>, <VALUE TYPE>, <MIN SIZE>, <MAX SIZE> ]
```

These parameters are optional. You must specify both key type and value type if you're going to specify one of them. The parameters must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

Position	Parameter	Data type	Default value	Description
1	Key type	Type	Scalar	What kinds of values can be used as keys. If you specify a key type, a value type is mandatory.
2	Value type	Type	Data	What kinds of values can be used as values.
3	Minimum size	Integer	0	The minimum number of key-value pairs in the hash. This parameter accepts the special value <code>default</code> , which uses its default value.
4	Maximum size	Integer	infinite	The maximum number of key-value pairs in the hash. This parameter accepts the special value <code>default</code> , which uses its default value.

Examples:

Hash

Matches a hash of any length; all keys must match `Scalar` and any values must match `Data`.

Hash[Integer, String]

Matches a hash that uses integers for keys and strings for values.

Hash[Integer, String, 1]

Same as previous, and requires a non-empty hash.

Hash[Integer, String, 1, 8]

Same as previous, and with a maximum size of eight key-value pairs.

The abstract `Struct` data type lets you specify the exact keys allowed in a hash, as well as what value types are allowed for each key.

Other abstract types, particularly `Variant` and `Enum`, are useful when specifying a value type for hashes that include multiple kinds of data.

Related information

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Abstract data types](#) on page 508

If you’re using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Numbers

Numbers in the Puppet language are normal integers and floating point numbers.

You can work with numbers using [arithmetic operators](#).

Numbers are written without quotation marks, and can consist only of:

- Digits.
- An optional negative sign (-). This is actually the unary negation operator rather than part of the number. Explicit positive signs (+) aren’t allowed.
- An optional decimal point, which results in a floating point value.
- An optional e or E for scientific notation of floating point values.
- An 0 prefix for octal base, or 0x or 0X prefix hexadecimal base.

Integers

Integers are numbers without decimal points.

If you divide two integers, the result is not a float. Instead, Puppet truncates the remainder. For example:

```
$my_number = 2 / 3      # evaluates to 0
$your_number = 5 / 3    # evaluates to 1
```

Floating point numbers

Floating point numbers (“floats”) are numbers that include a fractional value after a decimal point, including a fractional value of zero, as in `2.0`.

If an expression includes both integer and float values, the result is a float:

```
$some_number = 8 * -7.992      # evaluates to -63.936
$another_number = $some_number / 4 # evaluates to -15.984
```

Floating point numbers between -1 and 1 cannot start with a bare decimal point. They must have a zero before the decimal point:

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

You can express floating point numbers in scientific notation: append e or E, plus an exponent, and the preceding number is multiplied by 10 to the power of that exponent. Numbers in scientific notation are always floats:

```
$product = 8 * 3e5 # evaluates to 2400000.0
```

Octal and hexadecimal integers

Integer values can be expressed in decimal notation (base 10), octal notation (base 8), and hexadecimal notation (base 16).

Decimal (base 10) integers (other than 0) must not start with a 0.

Octal (base 8) integers have a prefix of 0 (zero), followed by octal digits 0 to 7.

Hexadecimal (base 16) integers have a prefix of 0x or 0X, followed by hexadecimal digits 0 to 9, a to f, or A to F.

Floats can't be expressed in octal or hexadecimal.

Examples:

```
# octal
$value = 0777    # evaluates to decimal 511
$value = 0789    # Error, invalid octal
$value = 0777.3  # Error, invalid octal

# hexadecimal
$value = 0x777   # evaluates to decimal 1911
$value = 0xdef   # evaluates to decimal 3567
$value = 0Xdef   # same as above
$value = 0xDEF   # same as above
$value = 0xLMN   # Error, invalid hex
```

Converting numbers to strings

Numbers are automatically converted to strings when interpolated into a string. The automatic conversion uses decimal (base 10) notation.

You can also [cast a number into a string](#) directly, by declaring a new `String` object.

Examples:

```
$from_integer = String(342)
$from_float   = String(3.14159)
```

To convert numbers to non-decimal string representations, use [the `sprintf` function](#).

Converting strings to numbers

Arithmetic operators in an expression automatically convert strings to numbers, but in all other contexts (for example, resource attributes or function arguments), Puppet won't automatically convert strings to numbers.

To convert a string to a number, [cast the type](#) by declaring a new `Numeric` object.

Examples:

```
$integer_var = Integer('342')
$float_var   = Float('3.14159')
$numeric_var = Numeric('5280')
```

For more information about casting, see the function documentation for [converting to Integer](#) and [converting to Float](#).

To extract numbers from strings, use [the `scanf` function](#). This function handles surrounding non-numerical text.

The Integer data type

The data type of integers is `Integer`. By default, `Integer` matches whole numbers of any size, within the limits of available memory. You can use parameters to restrict which values `Integer` matches.

Parameters

The full signature for `Integer` is:

```
Integer[<MIN VALUE>, <MAX VALUE>]
```

These parameters are optional. They must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

Position	Parameter	Data type	Default	Description
1	Minimum value	Integer	negative infinity	The minimum value for the integer. This parameter accepts the special value <code>default</code> , which uses its default value.
2	Maximum value	Integer	infinity	The maximum value for the integer. This parameter accepts the special value <code>default</code> , which uses its default value.

Practically speaking, the integer size limit is the range of a 64-bit signed integer (#9,223,372,036,854,775,808 to 9,223,372,036,854,775,807), which is the maximum size that can roundtrip safely between the components in the Puppet ecosystem.

Examples:

Integer

Matches any integer.

Integer[0]

Matches any integer greater than or equal to 0.

Integer[default, 0]

Matches any integer less than or equal to 0.

Integer[2, 8]

Matches any integer from 2 to 8, inclusive.

The Float data type

The data type of floating point numbers is `Float`. By default, `Float` matches floating point numbers within the limitations of Ruby's `Float` class. Practically speaking, this means a 64-bit double precision floating point value. You can use parameters to restrict which values `Float` matches.

Parameters

The full signature for `Float` is:

```
Float[<MIN VALUE>, <MAX VALUE>]
```

These parameters are optional. They must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

Position	Parameter	Data type	Default	Description
1	Minimum value	Float	negative infinity	The minimum value for the float. This parameter accepts the special value <code>default</code> , which uses its default value.
2	Maximum value	Float	infinity	The maximum value for the float. This parameter accepts the special value <code>default</code> , which uses its default value.

Examples:

Float

Matches any floating point number.

Float[1.6]

Matches any floating point number greater than or equal to 1.6.

Float[1.6, 3.501]

Matches any floating point number from 1.6 to 3.501, inclusive.

For more information about Float see Ruby's [Float class docs](#).

The Numeric data type

The data type of all numbers, both integer and floating point, is `Numeric`. It matches any integer or floating point number, and takes no parameters.

`Numeric` is equivalent to `Variant[Integer, Float]`. If you need to set size limits but still accept both integers and floats, you can use the abstract type `Variant` to construct an appropriate data type. For example:

```
Variant[Integer[-3,3], Float[-3.0,3.0]]
```

Related information

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Abstract data types](#) on page 508

If you’re using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Regular expressions

A regular expression (sometimes shortened to “regex” or “regexp”) is a pattern that can match some set of strings, and optionally capture parts of those strings for further use.

You can use regular expression values with the `=~` and `!~` match operators, case statements and selectors, node definitions, and functions like `regsubst` for editing strings, or `match` for capturing and extracting substrings. Regular expressions act like any other value, and can be assigned to variables and used in function arguments.

Syntax

Regular expressions are written as patterns enclosed within forward slashes. Unlike in Ruby, you cannot specify options or encodings after the final slash, like `/node .* /m`.

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

Puppet uses [Ruby's standard regular expression implementation](#) to match patterns. Other forms of regular expression quoting, like Ruby's `%r{^www(\d+)\.}`, are not allowed. You cannot interpolate variables or expressions into regex values.

If you are matching against a string that contains newlines, use `\A` and `\z` instead of `^` and `$`, which match the beginning and end of a line. This is a common mistake that can cause your regex to unintentionally match multiline text.

Some places in the language accept both real regex values and stringified regexes — that is, the same pattern quoted as a string instead of surrounded by slashes.

Regular expression options

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

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

The following options are available:

i

Ignore case.

m

Treat a new line as a character matched by `.`

x

Ignore whitespace and comments in the pattern.

Regular expression capture variables

Within [conditional statements](#) and [node definitions](#), substrings withing parentheses `()` in a regular expression are available as numbered variables inside the associated code section. The first is `$1`, the second is `$2`, and so on. The entire match is available as `$0`.

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

- The values of the numbered variables do not persist outside the code block associated with the pattern that set them.
- You can't manually assign values to a variable with only digits in its name; they can only be set by pattern matching.
- In nested conditionals, each conditional has its own set of values for the set of numbered variables. At the end of an interior statement, the numbered variables are reset to their previous values for the remainder of the outside statement. This causes conditional statements to act like [local scopes](#), but only with regard to the numbered variables.

The Regexp data type

The data type of regular expressions is `Regexp`. By default, `Regexp` matches any regular expression value. If you are looking for a type that matches strings which match arbitrary regular expressions, see the `Pattern` type. You can use parameters to restrict which values `Regexp` matches.

Parameters

The full signature for `Regexp` is:

```
Regexp[ <SPECIFIC REGULAR EXPRESSION> ]
```

The parameter is optional.

Position	Parameter	Data type	Default value	Description
1	Specific regular expression	<code>Regexp</code>	none	If specified, this results in a data type that only matches one specific regular expression value. Specifying a parameter here doesn't have a practical use.

Examples:

`Regexp`

Matches any regular expression.

`Regexp[/<regex> /]`

Matches the regular expression `/<regex>/` only.

`Regexp` matches only literal regular expression values. Don't confuse it with the abstract `Pattern` data type, which uses a regular expression to match a limited set of `String` values.

Related information

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Abstract data types](#) on page 508

If you're using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Resource and class references

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

The general form of a [resource](#) reference is:

- The resource type, capitalized. If the resource type includes a namespace separator `:`, then each segment must be capitalized.
- An opening square bracket `[`.
- The title of the resource as a string, or a comma-separated list of titles.
- A closing square bracket `]`.

For example, here is a reference to a file resource:

```
subscribe => File['/etc/ntp.conf'],
```

Here is a type with a multi-segment name:

```
before => Concat::Fragment['apache_port_header'],
```

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

Class references

Class references work the same as resource references, but use the pseudo-resource type `Class` instead of some other resource type name:

```
require => Class['ntp::install'],
```

Multi-resource references

Resource reference expressions with an array of titles or comma-separated list of titles refer to multiple resources of the same type. They evaluate to an array of single-title resource references. For example, here is a multi-resource reference:

```
require => File['/etc/apache2/httpd.conf', '/etc/apache2/magic', '/etc/
apache2/mime.types'],
```

And here is an equivalent multi-resource reference:

```
$my_files = ['/etc/apache2/httpd.conf', '/etc/apache2/magic', '/etc/apache2/
mime.types']
require => File[$my_files]
```

Multi-resource references can be used wherever an array of references can be used. They can go on either side of a [chaining arrow](#) or receive a [block of additional attributes](#).

Accessing attribute values

You can use a resource reference to access the values of a resource's attributes. To access a value, use square brackets and the name of an attribute (as a string). This works much like [accessing hash values](#).

```
file { "/etc/first.conf":
  ensure => file,
  mode   => "0644",
  owner  => "root",
}

file { "/etc/second.conf":
  ensure => file,
  mode   => File["/etc/first.conf"]["mode"],
  owner  => File["/etc/first.conf"]["owner"],
}
```

The resource whose values you're accessing must exist.

Like referencing variables, attribute access depends on evaluation order: Puppet must evaluate the resource you're accessing before you try to access it. If it hasn't been evaluated yet, Puppet raises an evaluation error.

You can only access attributes that are valid for that resource type. If you try to access a nonexistent attribute, Puppet raises an evaluation error.

Puppet can read the values of only those attributes that are explicitly set in the resource's declaration. It can't read the values of properties that would have to be read from the target system. It also can't read the values of attributes that default to some predictable value. For example, in the code above, you wouldn't be able to access the value of the `path` attribute, even though it defaults to the resource's title.

Like with hash access, the value of an attribute whose value was never set is `undef`.

Resource references as data types

If you've read the [Data type syntax](#) page, or perused the lower sections of the other data type pages, you might have noticed that resource references use the same syntax as values that represent data types. Internally, they're implemented the same way, and each resource reference is actually a data type.

For most users, this doesn't matter at all. Treat resource references as a special case with a coincidentally similar syntax, and it'll make your life generally easier. But if you're interested in the details, see [Resource types](#).

To restrict values for a class or defined type parameter so that users must provide your code a resource reference, do one of the following.

- To allow a resource reference of any resource type, use a data type of:

```
Type[Resource]
```

- To allow resource references *and* class references, use a data type of:

```
Type[Catalogentry]
```

- To allow a resource reference of a specific resource type — in this example, `file` — use one of the following:

```
Type[File]                # Capitalized resource type name
Type[Resource["file"]]    # Resource data type, with type name in parameter
                           as a string
Type[Resource[File]]      # Resource data type, with capitalized resource
                           type name
```

Any of these three options allow any `File['<TITLE>']` resource reference, while rejecting ones like `Service[<TITLE>]`.

Resource types

Resource types are a special family of data types that behave differently from other data types. They are subtypes of the fairly abstract `Resource` data type. Resource references are a useful subset of this data type family.

In the Puppet language, there are never any values whose data type is one of these resource types. That is, you can never create an expression where `$my_value =~ Resource` evaluates to `true`. For example, a [resource declaration](#) — an expression whose value you might expect would be a resource — executes a side effect and then produces a [resource reference](#) as its value. A resource reference is a data type in this family of data types, rather than a value that has one of these data types.

In almost all situations, if one of these resource type data types is involved, it makes more sense to treat it as a special language keyword than to treat it as part of a hierarchy of data types. It does have a place in that hierarchy; it's just complicated, and you don't need to know it to do things in the Puppet language.

For that reason, the information on this page is provided for the sake of technical completeness, but learning it isn't critical to your ability to use Puppet successfully.

Basics

Puppet automatically creates new known data type values for every resource type it knows about, including custom resource types and defined types.

These one-off data types share the name of the resource type they correspond to, with the first letter of every namespace segment capitalized. For example, the `file` type creates a data type called `File`.

Additionally, there is a parent `Resource` data type. All of these one-off data types are more-specific subtypes of `Resource`.

Usage of resource types without a title

A resource data type can be used in the following places:

- The resource type slot of a resource declaration.
- The resource type slot of a resource default statement.

For example:

```
# A resource declaration using a resource data type:
File { "/etc/ntp.conf":
  mode  => "0644",
  owner => "root",
  group => "root",
}

# Equivalent to the above:
Resource["file"] { "/etc/ntp.conf":
  mode  => "0644",
  owner => "root",
  group => "root",
}

# A resource default:
File {
  mode  => "0644",
  owner => "root",
  group => "root",
}
```

Usage of resource types with a title

If a resource data type includes a title, it acts as a resource reference, which are useful in several places. For more information, see [Resource and class references](#) on page 535.

The <SOME ARBITRARY RESOURCE TYPE> data type

For each resource type `mytype` known to Puppet, there is a data type, `Mytype`. It matches no values that can be produced in the Puppet language. You can use parameters to restrict which values `Mytype` matches, but it still matches no values.

Parameters

The full signature for a resource-type-corresponding data type `Mytype` is:

```
Mytype[ <RESOURCE TITLE> ]
```

This parameter is optional.

Position	Parameter	Data type	Default value	Description
1	Resource title	String	nothing	The title of some specific resource of this type. If provided, this turns this data type into a usable resource reference .

Examples:

File

The data type corresponding to the `file` resource type.

File[`'/tmp/filename.ext'`]

A resource reference to the `file` resource whose title is `/tmp/filename.ext`.

Type[File]

The data type that matches any [resource references](#) to `file` resources. This is useful for, for example, restricting the values of class or defined type parameters.

The Resource data type

There is also a general `Resource` data type, which all `<SOME ARBITRARY RESOURCE TYPE>` data types are more-specific subtypes of. Like the `Mytype`-style data types, it matches no values that can be produced in the Puppet language. You can use parameters to restrict which values `Resource` matches, but it still matches no values.

This is useful in the following uncommon circumstances:

- You need to interact with a resource type before you know its name. For example, you can do some clever business with the iteration functions to re-implement the `create_resources` function in the Puppet language, where your lambda receives arguments telling it to create resources of some resource type at runtime.
- Someone has somehow created a resource type whose name is invalid in the Puppet language, possibly by conflicting with a reserved word — you can use a `Resource` value to refer to that resource type in resource declarations and resource default statements, and to create resource references.

Parameters

The full signature for `Resource` is:

```
Resource[ <RESOURCE TYPE>, <RESOURCE TITLE>... ]
```

All of these parameters are optional. They must be listed in order; if you need to specify a later parameter, you must specify values for any prior ones.

Position	Parameter	Data type	Default value	Description
1	Resource type	String or <code>Resource</code>	nothing	A resource type, either as a string or a <code>Resource</code> data type value. If provided, this turns this data type into a resource-specific data type. <code>Resource[Mytype]</code> and <code>Resource["mytype"]</code> are identical to the data type <code>Mytype</code> .

Position	Parameter	Data type	Default value	Description
2 and higher	Resource title	String	nothing	The title of some specific resource of this type. If provided, this turns this data type into a usable resource reference or array of resource references. <code>Resource[Mytype, "mytitle"]</code> and <code>Resource["mytitle"]</code> are identical to the data type <code>Mytype["mytitle"]</code> .

Examples:

Resource[File]

The data type corresponding to the file resource type.

Resource[File, '/tmp/filename.ext']

A resource reference to the file resource whose title is `/tmp/filename.ext`.

Resource["file", '/tmp/filename.ext']

A resource reference to the file resource whose title is `/tmp/filename.ext`.

Resource[File, '/tmp/filename.ext', '/tmp/bar']

Equivalent to `[File['/tmp/filename.ext'], File['/tmp/bar']]`.

Type[Resource[File]]

A synonym for the data type that matches any resource references to file resources. This is useful for, for example, restricting the values of class or defined type parameters.

Type[Resource["file"]]

Another synonym for the data type that matches any resource references to file resources. This is useful for, for example, restricting the values of class or defined type parameters.

The Class data type

The `Class` data type is roughly equivalent to the set of `Mytype` data types, except it is for classes. Like the `Mytype`-style data types, it matches no values that can be produced in the Puppet language. You can use parameters to restrict which values `Class` matches, but it will matches no values.

Parameters

The full signature for `Class` is:

```
Class[<CLASS NAME>]
```

This parameter is optional.

Position	Parameter	Data type	Default value	Description
1	Class name	String	nothing	The name of a class. If provided, this turns this data type into a usable class reference .

Examples:

Class["apache"]

A [class reference](#) to class apache.

Type[Class]

The data type that matches any [class references](#). This is useful for, for example, restricting the values of class or defined type parameters.

Related data types

The abstract `Catalogentry` data type is the supertype of `Resource` and `Class`. You can use `Type[Catalogentry]` as the data type for a class or defined type parameter that can accept both class references and resource references.

Related information

[Resource and class references](#) on page 535

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

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Abstract data types](#) on page 508

If you’re using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Sensitive

Sensitive types in the Puppet language are strings marked as sensitive. The value is displayed in plain text in the catalog and manifest, but is redacted from logs and reports. Because the value is maintained as plain text, use it only as an aid to ensure that sensitive values are not inadvertently disclosed.

The `Sensitive` type can be written as `Sensitive.new(val)`, or the short form `Sensitive(val)`.

Parameters

The full signature for `Sensitive` is:

```
Sensitive.new([<STRING VALUE>])
```

The `Sensitive` type is parameterized, but the parameterized type (the type of the value it contains) only retains the basic type. Sensitive information about the length or details about the contained data value can otherwise be leaked.

It is therefore not possible to have detailed data types and expect that the data type match. For example, `Sensitive[Enum[red, blue, green]]` fails if a value of `Sensitive('red')` is given. When a sensitive type is used, the type parameter must be generic; in this example a `Sensitive[String]` instead would match `Sensitive('red')`.

Consider, for example, if you assign a sensitive value and call `notice`:

```
$secret = Sensitive('myPassword')
notice($secret)
```

The example manifest would log the following notice:

```
Notice: Scope(Class[main]): Sensitive [value redacted]
```

To gain access to the original data, use the `unwrap` function:

```
$secret = Sensitive('myPassword')
$processed = $secret.unwrap
```

```
notice $processed
```

Use `Sensitive` and `unwrap` only as an aid for logs and reports. The data is not encrypted.

Note: `Sensitive` puts no limit on the wrapped data and can wrap other types, but they do not fully reveal the type of that data when type checking because it reveals too much detail.

Strings

Strings are unstructured text fragments of any length. They're a common and useful data type.

Strings can interpolate other values, and can use escape sequences to represent characters that are inconvenient or impossible to write literally. You can access substrings of a string by numerical index.

There are four ways to write literal strings in the Puppet language:

- Bare words
- Single-quoted strings
- Double-quoted strings
- Heredocs

Each of these have slightly different behavior around syntax, interpolation features, and escape sequences.

Bare words

Puppet treats certain bare words — that is, runs of alphanumeric characters without surrounding quotation marks — as single-word strings. Bare word strings are most commonly used with resource attributes that accept a limited number of one-word values.

To be treated as a string, a bare word must:

- Begin with a lower case letter;
- Contain only letters, digits, hyphens (-), and underscores (_); and
- Not be a [reserved word](#).

For example, in the following code, `running` is a bare word string:

```
service { "ntp":
  ensure => running, # bare word string
}
```

Unquoted words that begin with upper case letters are interpreted as [data types](#) or [resource references](#), not strings.

Bare word strings can't interpolate values and can't use escape sequences.

Single-quoted strings

Multi-word strings can be surrounded by single quotation marks, `'like this'`.

For example:

```
if $autoupdate {
  notice('autoupdate parameter has been deprecated and replaced with
package_ensure. Set this to latest for the same behavior as autoupdate =>
true.')
}
```

Line breaks within the string are interpreted as literal line breaks.

Single-quoted strings can't interpolate values.

Escape sequences

The following escape sequences are available in single-quoted strings:

Sequence	Result
\\	Single backslash
\'	Literal single quotation mark

Within single quotation marks, if a backslash is followed by any character other than another backslash or a single quotation mark, Puppet treats it as a literal backslash.

To include a literal double backslash use a quadruple backslash.

To include a backslash at the very end of a single-quoted string, use a double backslash instead of a single backslash. For example: `path => 'C:\Program Files(x86)\\'`

Tip: A good habit is to always use two backslashes where you want the result to be one backslash. For example, `path => 'C:\\Program Files(x86)\\'`

Double-quoted strings

Strings can be surrounded by double quotation marks, "like this".

Line breaks within the string are interpreted as literal line breaks. You can also insert line breaks with `\n` (Unix-style) or `\r\n` (Windows-style).

Double-quoted strings can interpolate values. See [Interpolation information](#) below.

Escape sequences

The following escape sequences are available in double-quoted strings:

Sequence	Result
\\	Single backslash
\n	New line
\r	Carriage return
\t	Tab
\s	Space
\\$	Literal dollar sign (to prevent interpolation)
\uXXXX	Unicode character number XXXX (a four-digit hexadecimal number)
\u{XXXXXX}	Unicode character XXXXXX (a hexadecimal number between two and six digits)
\"	Literal double quotation mark
\'	Literal single quotation mark

Within double quotation marks, if a backslash is followed by any character other than those listed above (that is, a character that is not a recognized escape sequence), Puppet logs a warning: `Warning: Unrecognized escape sequence`, and treats it as a literal backslash.

Tip: A good habit is to always use two backslashes where you want the result to be one backslash.

Heredocs

Heredocs let you quote strings with more control over escaping, interpolation, and formatting. They're especially good for long strings with complicated content.

Example

```
$gitconfig = @( "GITCONFIG"/L)
```

```

[user]
  name = ${displayname}
  email = ${email}
[color]
  ui = true
[alias]
  lg = "log --pretty=format: '%C(yellow)%h%C(reset) %s \
%C(cyan)%cr%C(reset) %C(blue)%an%C(reset) %C(green)%d%C(reset)' --graph"
  wdiff = diff --word-diff=color --ignore-space-at-eol \
--word-diff-regex='[[:alnum:]]+|^[[:space:]][[:alnum:]]+'
[merge]
  defaultToUpstream = true
[push]
  default = upstream
| GITCONFIG

file { "${homedir}/.gitconfig":
  ensure => file,
  content => $gitconfig,
}

```

Syntax

To write a heredoc, you place a heredoc tag in a line of code. This tag acts as a literal string value, but the content of that string is read from the lines that follow it. The string ends when an end marker is reached.

The general form of a heredoc string is:

heredoc tag

```
@( "ENDTEXT" /<X> )
```

You can use a heredoc tag in Puppet code, anywhere a string value is accepted. In the above example, the heredoc tag `@("GITCONFIG" /L)` completes the line of Puppet code: `$gitconfig = .`

A heredoc tag starts with `@(` and ends with `)`.

Between those characters, the heredoc tag contains end text (see below) — text that is used to mark the end of the string. You can optionally surround this end text with double quotation marks to enable interpolation (see below). In the above example, the end text is `GITCONFIG`.

It also optionally contains escape switches (see below), which start with a slash `/`. In the above example, the heredoc tag has the escape switch `/L`.

the string

```
This is the
text that makes
up my string.
```

The content of the string starts on the next line, and can run over multiple lines. If you specified escape switches in the heredoc tag, the string can contain the enabled escape sequences.

In the above example, the string starts with `[user]` and ends with `default = upstream`. It uses the escape sequence `\` to add cosmetic line breaks, because the heredoc tag contained the `/L` escape switch.

end marker

```
| - ENDTEXT
```

On a line of its own, the end marker consists of:

- Optional indentation and a pipe character (|) to indicate how much indentation is stripped from the lines of the string (see below).
- An optional hyphen character (-), with any amount of space around it, to trim the final line break from the string (see below).
- The end text, repeating the same end text used in the heredoc tag, always without quotation marks.

In the above example, the end marker is |
GITCONFIG.

If a line of code includes more than one heredoc tag, Puppet reads all of those heredocs in order: the first one begins on the following line and continue until its end marker, the second one begins on the line immediately after the first end marker, and so on. Puppet won't start evaluating additional lines of Puppet code until it reaches the end marker for the final heredoc tag from the original line.

End text

The heredoc tag contains piece of text called the *end text*. When Puppet reaches a line that contains only that end text (plus optional formatting control), the string ends.

Both occurrences of the end text must match exactly, with the same capitalization and internal spacing.

End text can be any run of text that doesn't include line breaks, colons, slashes, or parentheses. It can include spaces, and can be mixed case. The following are all valid end text:

- EOT
- ...end...end...
- Verse 8 of The Raven

Tip: To help with code readability, make your end text stand out from the content of the heredoc string, for example, by making it all uppercase.

Enabling interpolation

By default, heredocs do not allow you to [interpolate values](#) into the string content. You can enable interpolation by double-quoting the end text in the opening heredoc tag. That is:

- An opening tag like @(EOT) won't allow interpolation.
- An opening tag like @("EOT") allows interpolation.

Note: If you enable interpolation, but the string has a dollar character (\$) that you want to be literal, not interpolated, you must enable the literal dollar sign escape switch (/ \$) in the heredoc tag: @("EOT" / \$). Then use the escape sequence \\$ to specify the literal dollar sign in the string. See the information on enabling escape sequences, below, for more details.

Enabling escape sequences

By default, heredocs have no escape sequences and every character is literal (except interpolated expressions, if enabled). To enable escape sequences, add switches to the heredoc tag.

To enable individual escape sequences, add a slash (/) and one or more switches. For example, to enable an escape sequence for dollar signs (\\$) and new lines (\n), add /\$n to the heredoc tag :

```
@( "EOT" / $n )
```

To enable all escape sequences, add a slash and no switches:

```
@( "EOT" / )
```

Use the following switches to enable escape sequences:

Switch to put in the heredoc tag	Escape sequence to use in the heredoc string	Result in the string value
(automatic)	\\	Single backslash. This switch is enabled when any other escape sequence is enabled.
n	\n	New line
r	\r	Carriage return
t	\t	Tab
s	\s	Space
\$	\\$	Literal dollar sign (to prevent interpolation)
u	\uXXXX or \u{XXXXXX}	Unicode character number XXXX (a four-digit hexadecimal number) or XXXXXX (a two- to six-digit hexadecimal number)
L	\<New line or carriage return>	Nothing. This lets you put line breaks in the heredoc source code that won't appear in the string value.

Note: A backslash that isn't part of an escape sequence is treated as a literal backslash. Unlike in double-quoted strings, this won't log a warning.

Tip: If a heredoc has escapes enabled, and includes several literal backslashes in a row, make sure each literal backslash is represented by the \\ escape sequence. So, for example, If you want the result to include a double backslash, use four backslashes.

Enabling syntax checking

To enable syntax checking of heredoc text, add the name of the syntax to the heredoc tag. For example:

```
@( END:pp )
@( END:epp )
@( END:json )
```

If Puppet has a syntax checker for the given syntax, it validates the heredoc text, but only if the heredoc is static text and does not contain any interpolations. If Puppet has no checker available for the given syntax, it silently ignores the syntax tag.

Syntax checking in heredocs is useful for validating syntax earlier, avoiding later failure.

By default, heredocs are treated as text unless otherwise specified in the heredoc tag.

Stripping indentation

To make your code easier to read, you can indent the content of a heredoc to separate it from the surrounding code. To strip this indentation from the resulting string value, put the same amount of indentation in front of the end marker and use a pipe character (|) to indicate the position of the first “real” character on each line.

```
$mytext = @(EOT)
  This block of text is
  visibly separated from
  everything around it.
  | EOT
```

If a line has less indentation than you’ve indicated with the pipe, Puppet strips any spaces it can without deleting non-space characters.

If a line has more indentation than you’ve indicated with the pipe, the excess spaces are included in the final string value.

Important: Indentation can include tab characters, but Puppet won’t convert tabs to spaces, so make sure you use the exact same sequence of space and tab characters on each line.

Suppressing literal line breaks

If you enable the `L` escape switch, you can end a line with a backslash (\) to exclude the following line break from the string value. This lets you break up long lines in your source code without adding unwanted literal line breaks to the resulting string value.

For example, Puppet would read this as a single line:

```
lg = "log --pretty=format:'%C(yellow)%h%C(reset) %s \
%C(cyan)%cr%C(reset) %C(blue)%an%C(reset) %C(green)%d%C(reset) ' --graph"
```

Suppressing the final line break

By default, heredocs end with a trailing line break, but you can exclude this line break from the final string. To suppress it, add a hyphen (-) to the end marker, before the end text, but after the indentation pipe if you used one. This works even if you don’t have the `L` escape switch enabled.

For example, Puppet would read this as a string with no line break at the end:

```
$mytext = @("EOT")
  This is too inconvenient for ${double} or ${single} quotes, but must be
  one line.
  |-EOT
```

Interpolation

Interpolation allows strings to contain expressions, which can be replaced with their values. You can interpolate any expression that resolves to a value, except for statement-style function calls. You can interpolate expressions in double-quoted strings, and in heredocs with interpolation enabled.

To interpolate an [expression](#), start with a dollar sign and wrap the expression in curly braces, as in: "String content \${<EXPRESSION>} more content".

The dollar sign doesn’t have to have a space in front of it. It can be placed directly after any other character, or at the beginning of the string.

An interpolated expression can include quote marks that would end the string if they occurred outside the interpolation token. For example: "<VirtualHost *:\${hiera("http_port")}>".

Preventing interpolation

If you want a string to include a literal sequence that looks like an interpolation token, but you don't want Puppet to try to evaluate it, use a quoting syntax that disables interpolation (single quotes or a non-interpolating heredoc), or escape the dollar sign with `\$`.

Short forms for variable interpolation

The most common thing to interpolate into a string is the value of a [variable](#). To make this easier, Puppet has some shorter forms of the interpolation syntax:

`$myvariable`

A variable reference (without curly braces) can be replaced with that variable's value. This also works with qualified variable names like `$myclass::myvariable`.

Because this syntax doesn't have an explicit stopping point (like a closing curly brace), Puppet assumes the variable name is everything between the dollar sign and the first character that couldn't legally be part of a variable name. (Or the end of the string, if that comes first.)

This means you can't use this style of interpolation when a value must run up against some other word-like text. And even in some cases where you can use this style, the following style can be clearer.

`${myvariable}`

A dollar sign followed by a variable name in curly braces can be replaced with that variable's value. This also works with qualified variable names like `${myclass::myvariable}`.

With this syntax, you can follow a variable name with any combination of [chained function calls](#) or [hash access](#) / [array access](#) / [substring access](#) expressions. For example:

```
"Using interface ${::interfaces.split(',') [3]} for broadcast"
```

However, this doesn't work if the variable's name overlaps with a language keyword. For example, if you had a variable called `$inherits`, you would have to use normal-style interpolation:

```
"Inheriting ${$inherits.upcase}."
```

Conversion of interpolated values

Puppet converts the value of any interpolated expression to a string using these rules:

Data type	Conversion
String	The contents of the string, with any quoting syntax removed.
Undef	An empty string.
Boolean	The string 'true' or 'false'.
Number	The number in decimal notation (base 10). For floats, the value can vary on different platforms. Use the sprintf function for more precise formatting.
Array	A pair of square brackets ([and]) containing the array's elements, separated by a comma and a space (,), with no trailing comma. Each element is converted to a string using these rules.
Hash	A pair of curly braces ({ and }) containing a <KEY> => <VALUE> string for each key-value pair, separated by a comma and a space (,), with no trailing comma. Each key and value is converted to a string using these rules.

Data type	Conversion
Regular expression	A stringified regular expression.
Resource reference or data type	The value as a string.

Line breaks

Quoted strings can continue over multiple lines, and line breaks are preserved as a literal part of the string. Heredocs let you suppress these line breaks if you use the `L` escape switch.

Puppet does not attempt to convert line breaks, so whatever type of line break is used in the file (LF for *nix or CRLF for Windows) is preserved. Use escape sequences to insert specific types of line breaks into strings:

- To insert a CRLF in a manifest file that uses *nix line endings, use the `\r\n` escape sequences in a double-quoted string, or a heredoc with those escapes enabled.
- To insert an LF in a manifest that uses Windows line endings, use the `\n` escape sequence in a double-quoted string, or a heredoc with that escape enabled.

Encoding

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

However, all strings must be valid UTF-8. Future versions of Puppet might impose restrictions on string encoding, and using only UTF-8 protects you in this event. Also, PuppetDB removes invalid UTF-8 characters when storing catalogs.

Accessing substrings

Access substrings of a string by specifying a numerical index inside square brackets. The index consists of one integer, optionally followed by a comma and a second integer, for example `$string[3]` or `$string[3,10]`.

The first number of the index is the start position. Positive numbers count from the start of the string, starting at 0. Negative numbers count back from the end of the string, starting at `-1`.

The second number of the index is the stop position. Positive numbers are lengths, counting forward from the start position. Negative numbers are absolute positions, counting back from the end of the string (starting at `-1`). If the second number is omitted, it defaults to 1 (resolving to a single character).

Examples:

```
$mystring = 'abcdef'
notice( $mystring[0] )      # resolves to 'a'
notice( $mystring[0,2] )    # resolves to 'ab'
notice( $mystring[1,2] )    # resolves to 'bc'
notice( $mystring[1,-2] )   # resolves to 'bcde'
notice( $mystring[-3,2] )   # resolves to 'de'
```

Text outside the actual range of the string is treated as an infinite amount of empty string:

```
$mystring = 'abcdef'
notice( $mystring[10] )     # resolves to ''
notice( $mystring[3,10] )   # resolves to 'def'
notice( $mystring[-10,2] )  # resolves to ''
notice( $mystring[-10,6] )  # resolves to 'ab'
```

The `String` data type

The data type of strings is `String`. By default, `String` matches strings of any length. You can use parameters to restrict which values `String` matches.

Parameters

The full signature for `String` is:

```
String[<MIN LENGTH>, <MAX LENGTH>]
```

These parameters are optional. They must be listed in order; if you need to specify a later parameter, you must also specify values for any prior ones.

Position	Parameter	Data type	Default value	Description
1	Minimum length	Integer	0	The minimum number of (Unicode) characters in the string. This parameter accepts the special value <code>default</code> , which uses its default value.
2	Maximum length	Integer	infinite	The maximum number of (Unicode) characters in the string. This parameter accepts the special value <code>default</code> , which uses its default value.

Examples:

`String`

Matches a string of any length.

`String[6]`

Matches a string with at least six characters.

`String[6,8]`

Matches a string with at least six and at most eight characters.

Related information

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Abstract data types](#) on page 508

If you’re using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Time-related data types

A `Timespan` defines the length of a duration of time, and a `Timestamp` defines a point in time. For example, “two hours” is a duration that can be represented as a `Timespan`, while “three o’clock in the afternoon UTC on 8 November, 2018” is a point in time that can be represented as a `Timestamp`. Both types can use nanosecond values if it is available on the platform.

The Timespan data type

A Timespan value represents a duration of time. The Timespan data type matches a specified range of durations and includes all values within the given range. The default represents a positive or negative infinite duration. A Timespan value can be specified with strings or numbers in various forms. The type takes up to two parameters.

Parameters

The full signature for Timespan is:

```
Timespan[ (<TIMESPAN START OR LENGTH>, (<TIMESPAN END>)) ]
```

Position	Parameter	Data type	Default value	Description
1	Timespan start or length	String, Float, Integer, or default	default (negative infinity in a span)	If only one parameter is passed, it is the length of the timespan. If two parameters are passed, this is the start or from value in a time range.
2	Timespan end	String, Float, Integer, or default	default (positive infinity) or none if only one value passed.	The end or to value in a time range.

Timespan values are interpreted depending on their format:

- A String in the format D-HH:MM:SS represents a duration of D days, HH hours, MM minutes, and SS seconds.
- An Integer or Float represents a duration in seconds.

A Timespan defined as a range (two parameters) matches any Timespan durations that can fit within that range. If either end of a range is defined as default (infinity), it is an *open range*, while any other range is a *closed range*. The range is inclusive.

In other words, Timespan[2] matches a duration of two seconds, but Timespan[0, 2] can match *any* Timespan from zero to two seconds, inclusive.

The Timespan type is not enumerable.

For information about converting values of other types to Timespan using the new function, or for converting a Timespan to a String using strftime, see the [function reference documentation](#).

Examples:

Timespan[2]

Matches a Timespan value of 2 seconds.

Timespan[77.3]

Matches a Timespan value of 1 minute, 17 seconds, and 300 milliseconds (77.3 seconds).

Timespan['1-00:00:00', '2-00:00:00']

Matches a closed range of Timespan values between 1 and 2 days.

The Timestamp data type

A Timestamp value represents a specific point in time. The Timestamp data type can be one single point or any point within a given range, depending on the number of specified parameters. Timestamp values that include a default parameter represents an infinite range of either positive or negative Timestamps. A Timestamp value can be specified with strings or numbers in various forms.

Parameters

The full signature for `Timestamp` is:

```
Timestamp[ (<TIMESTAMP VALUE>, (<RANGE LIMIT>)) ]
```

The type takes up to two parameters, and defaults to an infinite range to the past and future. A `Timestamp` with one parameter represents a single point in time, while two parameters represent a range of `Timestamps`, with the first parameter being the `from` value and the second being the `to` value.

Position	Parameter	Data type	Default value	Description
1	Timestamp value	String, Float, Integer, or default	default (negative infinity in a range)	Point in time if passed alone, or <code>from</code> value in a range if passed with a second parameter.
2	Range limit	String, Float, Integer, or default	default (positive infinity), or none if only one value is passed	The <code>to</code> value in a range.

A `Timestamp` that is defined as a single point in time (one parameter) matches exactly that point.

A `Timestamp` that is defined as a range (two parameters) matches any point in time within that range. If either end of a range is defined as default (infinity), it is an *open range*, while any other range is a *closed range*. The range is inclusive.

So, `Timestamp['2000-01-01T00:00:00.000']` matches 0:00 UTC on 1 January, 2000, while `Timestamp['2000-01-01T00:00:00.000' , '2001-01-01T00:00:00.000']` matches `Timestamp` values from that point in time to a point in time one year later, inclusive.

`Timestamp` values are interpreted depending on their format.

For information about converting values of other types to `Timestamp` using the new function, or for converting a `Timespan` to a `String` using the `strftime` function, see the [function reference documentation](#).

Examples:

`Timestamp['2000-01-01T00:00:00.000' , default]`

Matches an open range of `Timestamps` from the start of the 21st century to an infinite point in the future.

`Timestamp['2012-10-10']`

Matches the exact `Timestamp` 2012-10-10T00:00:00.0 UTC.

`Timestamp[default , 1433116800]`

Matches an open range of `Timestamps` from an infinite point in the past to 2015-06-01T00:00:00 UTC, here expressed as seconds since the Unix epoch.

`Timestamp['2010-01-01' , '2015-12-31T23:59:59.999999999']`

Matches a closed range of `Timestamps` between the start of 2010 and the end of 2015.

Undef

Puppet's `undef` value is roughly equivalent to `nil` in Ruby. It represents the absence of a value. If the `strict_variables` setting isn't enabled, variables which have never been declared have a value of `undef`.

The `undef` value is useful for testing whether a variable has been set. Also, you can use it to un-set resource attributes that have inherited values from a [resource default](#), causing the attribute to be unmanaged.

The only value in the `Undef` data type is the bare word `undef`.

Conversion

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

When interpolated into a string, `undef` is converted to the empty string.

The `Undef` data type

The data type of `undef` is `Undef`. It matches only the value `undef`, and takes no parameters.

Several abstract data types can match the `undef` value:

- The `Data` type matches `undef` in addition to several other data types.
- The `Any` type matches any value, including `undef`.
- The `Optional` type wraps one other data type, and returns a type that matches `undef` in addition to that type.
- The `Variant` type can accept the `Undef` type as a parameter, which makes the resulting data type match `undef`.
- The `NotUndef` type matches any value except `undef`.

Related information

[Data type syntax](#) on page 521

Each value in the Puppet language has a data type, like “string.” There is also a set of values *whose data type is “data type.”* These values represent the other data types. For example, the value `String` represents the data type of strings. The value that represents the data type of *these* values is `Type`.

[Abstract data types](#) on page 508

If you’re using data types to match or restrict values and need more flexibility than what the core data types (such as `String` or `Array`) allow, you can use one of the *abstract data types* to construct a data type that suits your needs and matches the values you want.

Templates

Templates are written in a specialized templating language that generates text from data. Use templates to manage the content of your Puppet configuration files via the `content` attribute of the `file` resource type.

Templating languages

Puppet supports two templating languages:

- Embedded Puppet (EPP) uses Puppet expressions in special tags. EPP works with Puppet 4.0 and later, and with Puppet 3.5 through 3.8 with future parser enabled.
- Embedded Ruby (ERB) uses Ruby code in tags, and requires some Ruby knowledge. ERB works with all Puppet versions.

When to use a template

Templates are more powerful than normal strings, and less powerful than modeling individual settings as resources. Whether to use a template is mainly a question of the complexity of the work you're performing.

When you're managing simple config files, a template generally isn't necessary because strings in the Puppet language allow you to interpolate variables and expressions into text. For short and simple config files, you can often use a [heredoc](#) and interpolate a few variables, or do something like `${ my_array.join(' , ') }`.

Use a template if you’re doing complex transformations (especially iterating over collections) or working with very large config files.

Some situations, however, are too complex for a template to be effective. For example, using several modules that each need to manage parts of the same config file is impractical with either templates or interpolated strings. For shared configuration like this, model each setting in the file as an individual resource, with either a custom resource type or an [Augeas](#), [concat](#), or [file_line](#) resource. This approach is similar to how core resource types like `ssh_authorized_key` and `mount` work.

- [Creating templates using Embedded Puppet](#) on page 554

Embedded Puppet (EPP) is a templating language based on the Puppet language. You can use EPP in Puppet 4 and higher, and with Puppet 3.5 through 3.8 with the future parser enabled. Puppet evaluates EPP templates with the `epp` and `inline_epp` functions.

- [Creating templates using Embedded Ruby](#) on page 561

Embedded Ruby (ERB) is a templating language based on Ruby. Puppet evaluates ERB templates with the `template` and `inline_template` functions.

Creating templates using Embedded Puppet

Embedded Puppet (EPP) is a templating language based on the Puppet language. You can use EPP in Puppet 4 and higher, and with Puppet 3.5 through 3.8 with the future parser enabled. Puppet evaluates EPP templates with the `epp` and `inline_epp` functions.

EPP structure and syntax

An EPP template looks like a plain-text document interspersed with tags containing Puppet expressions. When evaluated, these tagged expressions can modify text in the template. You can use Puppet variables in an EPP template to customize its output.

The following example shows parameter tags (`<% |`), non-printing expression tags (`<%`), expression-printing tags (`<%=`), and comment tags (`<%#`). A hyphen in a tag (`-`) strips leading or trailing whitespace when printing the evaluated template:

```
<%- | Boolean $keys_enable,
      String $keys_file,
      Array  $keys_trusted,
      String $keys_requestkey,
      String $keys_controlkey
| -%>
<% if $keys_enable { -%>

<%# Printing the keys file, trusted key, request key, and control key: -%>
keys <%= $keys_file %>
<% unless $keys_trusted =~ Array[Data,0,0] { -%>
trustedkey <%= $keys_trusted.join(' ') %>
<% } -%>
<% if $keys_requestkey =~ String[1] { -%>
requestkey <%= $keys_requestkey %>
<% } -%>
<% if $keys_controlkey =~ String[1] { -%>
controlkey <%= $keys_controlkey %>
<% } -%>

<% } -%>
```

EPP tags

Embedded Puppet (EPP) has two tags for Puppet code expressions, optional tags for parameters and comments, and a way to escape tag delimiters.

The following table provides an overview of the main tag types used with EPP. See the sections below for additional detail about each tag, including instructions on trimming whitespace and escaping special characters.

I want to ...	EPP tag syntax
Insert the value of a single expression.	<code><%= EXPRESSION %></code>
Execute an expression without inserting a value.	<code><% EXPRESSION %></code>
Declare the template's parameters.	<code><% PARAMETERS %></code>
Add a comment.	<code><%# COMMENT %></code>

Text outside a tag is treated as literal text, but is subject to any tagged Puppet code surrounding it. For example, text surrounded by a tagged `if` statement only appears in the output if the condition is true.

Expression-printing tags

An expression-printing tag inserts the value of a single Puppet expression into the output.

Opening tag	<code><%=</code>
Closing tag	<code>%></code>
Closing tag with trailing whitespace and line break trimming	<code>-%></code>

Example tag:

```
<%= $fqdn %>
```

An expression-printing tag must contain any single Puppet expression that resolves to a value, including plain variables, [function calls](#), and arithmetic expressions. If the value isn't a string, Puppet automatically converts it to a string based on the [rules for value interpolation in double-quoted strings](#).

All facts are available in EPP templates. For example, to insert the value of the `fqdn` and `hostname` facts in an EPP template for an Apache config file:

```
ServerName <%= $facts[fqdn] %>
ServerAlias <%= $facts[hostname] %>
```

Non-printing tags

A non-printing tag executes the code it contains, but doesn't insert a value into the output.

Opening tag	<code><%</code>
Opening tag with indentation trimming	<code><%-</code>
Closing tag	<code>%></code>
Closing tag with trailing whitespace and line break trimming	<code>-%></code>

Non-printing tags that contain [iterative](#) and [conditional](#) expressions can affect the untagged text they surround.

For example, to insert text only if a certain variable was set, write:

```
<% if $broadcastclient == true { -%>
broadcastclient
<% } -%>
```

Expressions in non-printing tags don't have to resolve to a value or be a complete statement, but the tag must close at a place where it would be legal to write another expression. For example, this doesn't work:

```
<%# Syntax error: %>
<% $servers.each -%>
# some server
<% |$server| { %> server <%= server %>
<% } -%>
```

You must keep `|$server| {` inside the first tag, because you can't insert an arbitrary statement between a function call and its required block.

Parameter tags

A parameter tag declares which parameters the template accepts. Each parameter can be [typed](#) and can have a default value.

Opening tag with indentation trimming	<%-
Closing tag with trailing whitespace and line break trimming	-%>

Example tag:

```
<%- | Boolean $keys_enable = false, String $keys_file = '' | -%>
```

The parameter tag is optional; if used, it must be the first content in a template. Always close the parameter tag with a right-trimmed delimiter (`-%>`) to avoid outputting a blank line. Literal text, line breaks, and non-comment tags cannot precede the template's parameter tag. (Comment tags that precede a parameter tag must use the right-trimming tag to trim trailing whitespace.)

The parameter tag's pair of pipe characters (`|`) contains a comma-separated list of parameters. Each parameter follows this format:

```
Boolean $keys_enable = false
```

- An optional [data type](#), which restricts the allowed values for the parameter (defaults to Any)
- A [variable name](#)
- An optional equals (=) sign and default value, which must match the data type, if one was specified

Parameters with default values are optional, and can be omitted when the template is evaluated. If you want to use a default value of `undef`, make sure to also specify a data type that allows `undef`. For example, `Optional[String]` accepts `undef` as well as any string.

Comment tags

A comment tag's contents do not appear in the template's output.

Opening tag	<%#
Closing tag	%>
Closing tag with space trimming	-%>

Example tag:

```
<%# This is a comment. %>
```

Literal tag delimiters

If you need the template's final output to contain a literal `<%` or `%>`, you can escape the characters as `<%%` or `%%>`. The first literal tag is taken, and the rest of the line is treated as a literal. This means that `<%% Test %%>` in an EPP template would turn out as `<% Test %%>`, not `<% Test %>`.

Accessing EPP variables

Embedded Puppet (EPP) templates can access variables with the `$variable` syntax used in Puppet.

A template works like a [defined type](#):

- It has its own anonymous [local scope](#).
- The parent scope is set to node scope (or top scope if there's no [node definition](#)).
- When you call the template (with the `epp` or `inline_epp` functions), you can use parameters to set variables in its local scope.

- Unlike Embedded Ruby (ERB) templates, EPP templates cannot directly access variables in the calling class without namespacing. Fully qualify variables or pass them in as parameters.

EPP templates can use short names to access global variables (like `$os` or `$trusted`) and their own local variables, but must use qualified names (like `$ntp::tinker`) to access variables from any class. The exception to this rule is `inline_epp`.

Special scope rule for `inline_epp`

If you evaluate a template with the `inline_epp` function, and if the template has no parameters, either passed or declared, you can access variables from the calling class in the template by using the variables' short names. This exceptional behavior is only allowed if **all** of the above conditions are true.

Should I use a parameter or a class variable?

Templates have two ways to use data:

- Directly access class variables, such as `$ntp::tinker`
- Use parameters passed at call time

Use class variables when a template is closely tied to the class that uses it, you don't expect it to be used anywhere else, and you need to use a **lot** of variables.

Use parameters when a template is used in several different places and you want to keep it flexible. Remember that declaring parameters with a tag makes a template's data requirements visible at a glance.

EPP parameters

When you pass parameters when you call a template, the parameters become local variables inside the template. To use a parameter in this way, pass a [hash](#) as the last argument of the `epp` or `inline_epp` functions.

For example, calling this:

```
epp('example/example.epp', { 'logfile' => "/var/log/ntp.log" })
```

to evaluate this template:

```
<%- | Optional[String] $logfile = undef | -%>
<%# (Declare the $logfile parameter as optional) -%>

<% unless $logfile =~ Undef { -%>
logfile <%= $logfile %>
<% } -%>
```

The keys of the hash match the variable names you'll be using in the template, minus the leading `$` sign. Parameters must follow the normal rules for [local variable names](#).

If the template uses a parameter tag, it **must** be the first content in a template and you can **only** pass the parameters it declares. Passing any additional parameters is a syntax error. However, if a template omits the parameter tag, you can pass it any parameters.

If a template's parameter tag includes any parameters without default values, they are mandatory. You must pass values for them when calling the template.

Sensitive data

Since Puppet 6.20, Puppet can interpolate sensitive values. In your EPP template, you can access the sensitive variable without unwrapping. For example:

```
host=<%= scope['db_host'] %>
password=<%= scope['sensitive::db_password'] %>
```

The rendered output is automatically sensitive and used as the file content:

```
file { ['/etc/service.conf':
  ensure => file,
  content => epp('<module>/service.conf.erb')
}
```

Note: ERB templates do not support interpolation of sensitive values — you have to manually unwrap and re-wrap these.

Example EPP template

The following example is an EPP translation of the `ntp.conf.erb` template from the `puppetlabs-ntp` module.

```
# ntp.conf: Managed by puppet.
#
<% if $ntp::tinker == true and ($ntp::panic or $ntp::stepout) { -%>
# Enable next tinker options:
# panic - keep ntpd from panicking in the event of a large clock skew
# when a VM guest is suspended and resumed;
# stepout - allow ntpd change offset faster
tinker<% if $ntp::panic { %> panic <%= $ntp::panic %><% } %><% if
  $ntp::stepout { -%> stepout <%= $ntp::stepout %><% } %>
<% } -%>

<% if $ntp::disable_monitor == true { -%>
disable monitor
<% } -%>
<% if $ntp::disable_auth == true { -%>
disable auth
<% } -%>

<% if $ntp::restrict =~ Array[Data,1] { -%>
# Permit time synchronization with our time source, but do not
# permit the source to query or modify the service on this system.
<% $ntp::restrict.flatten.each |$restrict| { -%>
restrict <%= $restrict %>
<% } -%>
<% } -%>

<% if $ntp::interfaces =~ Array[Data,1] { -%>
# Ignore wildcard interface and only listen on the following specified
# interfaces
interface ignore wildcard
<% $ntp::interfaces.flatten.each |$interface| { -%>
interface listen <%= $interface %>
<% } -%>
<% } -%>

<% if $ntp::broadcastclient == true { -%>
broadcastclient
<% } -%>

# Set up servers for ntpd with next options:
# server - IP address or DNS name of upstream NTP server
# iburst - allow send sync packages faster if upstream unavailable
# prefer - select preferable server
# minpoll - set minimal update frequency
# maxpoll - set maximal update frequency
<% [$ntp::servers].flatten.each |$server| { -%>
server <%= $server %><% if $ntp::iburst_enable == true { %> iburst<% } %><%
  if $server in $ntp::preferred_servers { %> prefer<% } %><% if $ntp::minpoll
```

```

    { %> minpoll <%= $ntp::minpoll %><% } %><% if $ntp::maxpoll { %> maxpoll <
    %= $ntp::maxpoll %><% } %>
    <% } -%>

    <% if $ntp::udlc { -%>
    # Undisciplined Local Clock. This is a fake driver intended for backup
    # and when no outside source of synchronized time is available.
    server    127.127.1.0
    fudge     127.127.1.0 stratum <%= $ntp::udlc_stratum %>
    restrict 127.127.1.0
    <% } -%>

    # Driftfile.
    driftfile <%= $ntp::driftfile %>

    <% unless $ntp::logfile =~ Undef { -%>
    # Logfile
    logfile <%= $ntp::logfile %>
    <% } -%>

    <% unless $ntp::peers =~ Array[Data,0,0] { -%>
    # Peers
    <% [$ntp::peers].flatten.each |$peer| { -%>
    peer <%= $peer %>
    <% } -%>
    <% } -%>

    <% if $ntp::keys_enable { -%>
    keys <%= $ntp::keys_file %>
    <% unless $ntp::keys_trusted =~ Array[Data,0,0] { -%>
    trustedkey <%= $ntp::keys_trusted.join(' ') %>
    <% } -%>
    <% if $ntp::keys_requestkey =~ String[1] { -%>
    requestkey <%= $ntp::keys_requestkey %>
    <% } -%>
    <% if $ntp::keys_controlkey =~ String[1] { -%>
    controlkey <%= $ntp::keys_controlkey %>
    <% } -%>

    <% } -%>
    <% [$ntp::fudge].flatten.each |$entry| { -%>
    fudge <%= $entry %>
    <% } -%>

    <% unless $ntp::leapfile =~ Undef { -%>
    # Leapfile
    leapfile <%= $ntp::leapfile %>
    <% } -%>

```

To call this template from a manifest (assuming that the template file is located in the templates directory of the puppetlabs-ntp module), add the following code to the manifest:

```

# epp(<FILE REFERENCE>, [<PARAMETER HASH>])
file { '/etc/ntp.conf':
  ensure => file,
  content => epp('ntp/ntp.conf.epp'),
  # Loads /etc/puppetlabs/code/environments/production/modules/ntp/
  templates/ntp.conf.epp
}

```

Validating and previewing EPP templates

Before deploying a template, validate its syntax and render its output to make sure the template is producing the results you expect. Use the `puppet epp` command-line tool for validating and rendering Embedded Puppet (EPP) templates.

EPP validation

To validate your template, run: `puppet epp validate <TEMPLATE NAME>`

The `puppet epp` command includes an action that checks EPP code for syntax problems. The `<TEMPLATE NAME>` can be a file reference or can refer to a `<MODULE NAME>/<TEMPLATE FILENAME>` as the `epp` function. If a file reference can also refer to a module, Puppet validates the module's template instead.

You can also pipe EPP code directly to the validator: `cat example.epp | puppet epp validate`

The command is silent on a successful validation. It reports and halts on the first error it encounters. For information on how to modify this default behavior, see the command's [man page](#).

EPP rendering

To render your template, run: `puppet epp render <TEMPLATE NAME>`

You can render EPP from the command line with `puppet epp render`. If Puppet can evaluate the template, it outputs the result.

If your template relies on specific parameters or values to function, you can simulate those values by passing a hash to the `--values` option. For example:

```
puppet epp render example.epp --values '{x => 10, y => 20}'
```

You can also render inline EPP by using the `-e` flag or piping EPP code to `puppet epp render`, and even simulate facts using YAML. For details, see the command's [man page](#).

Evaluating EPP templates

After you have an EPP template, you can pass it to a function that evaluates it and returns a final string. The actual template can be either a separate file or a string value.

Evaluating EPP templates that are in a template file

Put template files in the `templates` directory of a module. EPP files use the `.epp` extension.

To use a EPP template file, evaluate it with the `epp` function. For example:

```
# epp(<FILE REFERENCE>, [<PARAMETER HASH>])
file { '/etc/ntp.conf':
  ensure => file,
  content => epp('ntp/ntp.conf.epp', {'service_name' => 'xntpd',
  'iburst_enable' => true}),
  # Loads /etc/puppetlabs/code/environments/production/modules/ntp/
  templates/ntp.conf.epp
}
```

The first argument to the function is the *file reference*: a string in the form `'<MODULE>/<FILE>'`, which loads `<FILE>` from `<MODULE>`'s `templates` directory. For example, the file reference `ntp/ntp.conf.epp` loads the `<MODULES DIRECTORY>/ntp/templates/ntp.conf.epp` file.

Some EPP templates declare parameters, and you can provide values for them by passing a *parameter hash* to the `epp` function.

The keys of the [hash](#) must be [valid local variable names](#) (minus the `$`). Inside the template, Puppet creates variables with those names and assign their values from the hash. For example, with a parameter hash

of `{'service_name' => 'xntpd', 'iburst_enable' => true}`, an EPP template would receive variables called `$service_name` and `$iburst_enable`.

When structuring your parameter hash, remember:

- If a template declares any mandatory parameters, you **must** set values for them with a parameter hash.
- If a template declares any optional parameters, you can choose to provide values or let them use their defaults.
- If a template declares **no** parameters, you can pass any number of parameters with any names; otherwise, you can only choose from the parameters requested by the template.

Evaluating EPP template strings

If you have a string value that contains template content, you can evaluate it with the `inline_epp` function.

In older versions of Puppet, inline templates were mostly used to get around limitations — tiny Ruby fragments were useful for transforming and manipulating data before Puppet had [iteration functions](#) like `map` or [puppetlabs/stdlib](#) functions like `chomp` and `keys`.

In modern versions of Puppet, inline templates are usable in some of the same situations template files are. Because the [heredoc](#) syntax makes it easy to write large and complicated strings in a manifest, you can use `inline_epp` to reduce the number of files needed for a simple module that manages a small config file.

For example:

```
$ntp_conf_template = @(END)
...template content goes here...
END

# inline_epp(<TEMPLATE STRING>, [<PARAMETER HASH>])
file { ['/etc/ntp.conf':
  ensure => file,
  content => inline_epp($ntp_conf_template, {'service_name' => 'xntpd',
    'iburst_enable' => true}),
}
```

Some EPP templates declare parameters, and you can provide values for them by passing a *parameter hash* to the `epp` function.

The keys of the [hash](#) must be [valid local variable names](#) (minus the `$`). Inside the template, Puppet creates variables with those names and assign their values from the hash. For example, with a parameter hash of `{'service_name' => 'xntpd', 'iburst_enable' => true}`, an EPP template would receive variables called `$service_name` and `$iburst_enable`.

When structuring your parameter hash, remember:

- If a template declares any mandatory parameters, you **must** set values for them with a parameter hash.
- If a template declares any optional parameters, you can choose to provide values or let them use their defaults.
- If a template declares **no** parameters, you can pass any number of parameters with any names; otherwise, you can only choose from the parameters requested by the template.

Creating templates using Embedded Ruby

Embedded Ruby (ERB) is a templating language based on Ruby. Puppet evaluates ERB templates with the `template` and `inline_template` functions.

If you've used [ERB](#) in other projects, it might have had different features enabled. This page describes how ERB works with Puppet.

Note: Puppet has a parallel templating system called Embedded Puppet (EPP), which has similar functionality to ERB, but is based on the Puppet language. EPP is the preferred and safer method because it isolates environments. See [EPP](#) and [environment isolation](#) for more information.

ERB structure and syntax

An ERB template looks like a plain-text document interspersed with tags containing Ruby code. When evaluated, this tagged code can modify text in the template.

Puppet passes data to templates via special objects and variables, which you can use in the tagged Ruby code to control the templates' output. The following example shows non-printing tags (<%), expression-printing tags (<%=), and comment tags (<%#). A hyphen in a tag (-) strips leading or trailing whitespace when printing the evaluated template:

```
<% if @keys_enable -%>
<%# Printing the keys file, trusted key, request key, and control key: -%>
keys <%= @keys_file %>
<% unless @keys_trusted.empty? -%>
trustedkey <%= @keys_trusted.join(' ') %>
<% end -%>
<% if @keys_requestkey != '' -%>
requestkey <%= @keys_requestkey %>
<% end -%>
<% if @keys_controlkey != '' -%>
controlkey <%= @keys_controlkey %>
<% end -%>

<% end -%>
```

ERB tags

Embedded Ruby (ERB) has two tags for Ruby code expressions, a tag for comments, and a way to escape tag delimiters.

The following table provides an overview of the main tag types used with ERB. See the sections below for additional detail about each tag, including instructions on trimming whitespace and escaping special characters.

I want to ...	ERB tag syntax
Insert the value of a single expression.	<%= EXPRESSION %>
Execute an expression without inserting a value.	<% EXPRESSION %>
Add a comment.	<%# COMMENT %>

Text outside a tag is treated as literal text, but is subject to any tagged Ruby code surrounding it. For example, text surrounded by a tagged `if` statement only appears in the output if the condition is true.

Expression-printing tags

An expression-printing tag inserts the value into the output.

Opening tag	<%=
Closing tag	%>
Closing tag with trailing whitespace and line break trimming	-%>

Example tag:

```
<%= @fqdn %>
```

It must contain a snippet of Ruby code that resolves to a value; if the value isn't a string, it is automatically converted to a string using its `to_s` method.

For example, to insert the value of the `fqdn` and `hostname` facts in an ERB template for an Apache config file:

```
ServerName <%= @fqdn %>
ServerAlias <%= @hostname %>
```

Non-printing tags

A non-printing tag executes the code it contains, but doesn't insert a value into the output.

Opening tag	<%
Opening tag with indentation trimming	<% -
Closing tag	%>
Closing tag with trailing whitespace and line break trimming	-%>

Non-printing tags that contain [iterative](#) and [conditional](#) expressions can affect the untagged text they surround.

For example, to insert text only if a certain variable was set, write:

```
<% if @broadcastclient == true -%>
broadcastclient
<% end -%>
```

Expressions in non-printing tags don't have to resolve to a value or be a complete statement, but the tag must close at a place where it would be legal to write another expression. For example, this doesn't work:

```
<%# Syntax error: %>
<% @servers.each -%>
# some server
<% do |server| %>server <%= server %>
<% end -%>
```

You must keep `do |server|` inside the first tag, because you can't insert an arbitrary statement between a function call and its required block.

Comment tags

A comment tag's contents do not appear in the template's output.

Opening tag	<%#
Closing tag	%>
Closing tag with line break trimming	-%>

Example tag:

```
<%# This is a comment. %>
```

Literal tag delimiters

If you need the template's final output to contain a literal `<%` or `%>`, you can escape the characters as `<%%` or `%%>`. The first literal tag is taken, and the rest of the line is treated as a literal. This means that `<%% Test %>` in an ERB template would turn out as `<% Test %>`, not `<% Test %>`.

Accessing Puppet variables

ERB templates can access Puppet variables. This is the main source of data for templates.

An ERB template has its own [local scope](#), and its parent scope is set to the class or defined type that evaluates the template. This means a template can use short names for variables from that class or type, but it can't insert new variables into it.

There are two ways to access variables in an ERB template:

- `@variable`
- `scope['variable']` and its older equivalent, `scope.lookupvar('variable')`

@variable

All variables in the current scope (including global variables) are passed to templates as Ruby instance variables, which begin with “at” signs (@). If you can access a variable by its short name in the surrounding manifest, you can access it in the template by replacing its \$ sign with an @, so that `$os` becomes `@os`, and `$trusted` becomes `@trusted`.

This is the most legible way to access variables, but it doesn't support variables from other scopes. For that, you need to use the `scope` object.

scope['variable'] or scope.lookupvar('variable')

Puppet also passes templates an object called `scope`, which can access all variables (including out-of-scope variables) with a hash-style access expression. For example, to access `$ntp::tinker` you would use `scope['ntp::tinker']`.

Another way to use the `scope` object is to call its `lookupvar` method and pass the variable's name as its argument, as in `scope.lookupvar('ntp::tinker')`. This is exactly equivalent to the above, if slightly less convenient. This usage predates the hash-style indexing added in Puppet 3.0.

Puppet data types in Ruby

Puppet's [data types](#) are converted to Ruby classes as follows:

Puppet type	Ruby class
Boolean	<code>Boolean</code>
String	<code>String</code>
Number	Subtype of <code>Numeric</code>
Array	<code>Array</code>
Hash	<code>Hash</code>
Default	<code>Symbol (value :default)</code>
Regex	<code>Regexp</code>
Resource reference	<code>Puppet::Pops::Types::PResourceType</code> or <code>Puppet::Pops::Types::PHostClassType</code>
Lambda (code block)	<code>Puppet::Pops::Evaluator::Closure</code>
Data type (type)	A type class under <code>Puppet::Pops::Types</code> , such as <code>Puppet::Pops::Types::PIntegerType</code>
Undef	<code>NilClass (value nil)</code>
	Note: If a Puppet variable was never defined, its value is <code>undef</code> , which means its value in a template is <code>nil</code> .

Using Ruby in ERB templates

To manipulate and print data in ERB templates, you'll need to know some Ruby. A full introductory Ruby tutorial is outside the scope of these docs, but this page provides an overview of Ruby basics commonly used in ERB templates.

Using if statements

The `if ... end` statement in Ruby lets you write conditional text. Put the control statements in non-printing tags, and the conditional text between the tags:

```
<% if <CONDITION> %> text goes here <% end %>
```

For example:

```
<% if @broadcast != "NONE" %>broadcast <%= @broadcast %><% end %>
```

The general format of an `if` statement is:

```
if <CONDITION>
  ... code ...
elsif <CONDITION>
  ... other code ...
end
```

Using iteration

Ruby lets you iterate over arrays and hashes with the `each` method. This method takes a block of code and executes it one time for each element in the array or hash. In a template, untagged text is treated as part of the code that gets repeated. You can think of literal text as an instruction, telling the evaluator to insert that text into the final output.

To write a block of code in Ruby, use either `do |arguments| ... end` or `{ |arguments| ... }`. Note that this is different from Puppet [lambdas](#) — but they work similarly.

```
<% @values.each do |val| -%>
Some stuff with <%= val %>
<% end -%>
```

If `$values` was set to `['one', 'two']`, this example would produce:

```
Some stuff with one
Some stuff with two
```

This example also trims line breaks for the non-printing tags, so they won't appear as blank lines in the output.

Manipulating data

Your templates generally use data from Puppet variables. These values almost always be [strings](#), [numbers](#), [arrays](#), and [hashes](#).

These become the equivalent Ruby objects when you access them from an ERB template.

For information about the ways you can transform these objects, see the Ruby documentation for [strings](#), [integers](#), [arrays](#), and [hashes](#).

Also, note that the [special undef value](#) in Puppet becomes the special `nil` value in Ruby in ERB templates.

Calling Puppet functions from ERB templates

You can use Puppet functions inside ERB templates by calling the `scope.call_function(<NAME>, <ARGS>)` method.

This method takes two arguments:

- The name of the function, as a string.
- All arguments to the function, as an array. This must be an array even for one argument or zero arguments.

For example, to evaluate one template inside another:

```
<%= scope.call_function('template', ["my_module/template2.erb"]) %>
```

To log a warning using the Puppet logging system, so that the warning appears in reports:

```
<%= scope.call_function('warning', ["Template was missing some data; this
  config file might be malformed."]) %>
```

Note:

`scope.call_function` was added in Puppet 4.2.

Previous versions of Puppet created a `function_<NAME>` method on the `scope` object for each function. These could be called with an arguments array, such as `<%= scope.function_template(["my_module/template2.erb"]) %>`.

While this method still works in Puppet 4.2 and later versions, the auto-generated methods don't support the modern function APIs, which are now used by the majority of built-in functions.

Example ERB template

The following example is taken from the `puppetlabs-ntp` module.

```
# ntp.conf: Managed by puppet.
#
<% if @tinker == true and (@panic or @stepout) -%>
# Enable next tinker options:
# panic - keep ntpd from panicking in the event of a large clock skew
# when a VM guest is suspended and resumed;
# stepout - allow ntpd change offset faster
tinker<% if @panic -%> panic <%= @panic %><% end %><% if @stepout -%>
  stepout <%= @stepout %><% end %>
<% end -%>

<% if @disable_monitor == true -%>
disable monitor
<% end -%>
<% if @disable_auth == true -%>
disable auth
<% end -%>

<% if @restrict != [] -%>
# Permit time synchronization with our time source, but do not
# permit the source to query or modify the service on this system.
<% @restrict.flatten.each do |restrict| -%>
restrict <%= restrict %>
<% end -%>
<% end -%>

<% if @interfaces != [] -%>
# Ignore wildcard interface and only listen on the following specified
# interfaces
interface ignore wildcard
<% @interfaces.flatten.each do |interface| -%>
interface listen <%= interface %>
<% end -%>
<% end -%>

<% if @broadcastclient == true -%>
broadcastclient
```

```

<% end -%>

# Set up servers for ntpd with next options:
# server - IP address or DNS name of upstream NTP server
# iburst - allow send sync packages faster if upstream unavailable
# prefer - select preferrable server
# minpoll - set minimal update frequency
# maxpoll - set maximal update frequency
<% [@servers].flatten.each do |server| -%>
server <%= server %><% if @iburst_enable == true -%> iburst<% end %><% if
  @preferred_servers.include?(server) -%> prefer<% end %><% if @minpoll -%>
  minpoll <%= @minpoll %><% end %><% if @maxpoll -%> maxpoll <%= @maxpoll %><
% end %>
<% end -%>

<% if @udlc -%>
# Undisciplined Local Clock. This is a fake driver intended for backup
# and when no outside source of synchronized time is available.
server 127.127.1.0
fudge 127.127.1.0 stratum <%= @udlc_stratum %>
restrict 127.127.1.0
<% end -%>

# Driftfile.
driftfile <%= @driftfile %>

<% unless @logfile.nil? -%>
# Logfile
logfile <%= @logfile %>
<% end -%>

<% unless @peers.empty? -%>
# Peers
<% [@peers].flatten.each do |peer| -%>
peer <%= peer %>
<% end -%>
<% end -%>

<% if @keys_enable -%>
keys <%= @keys_file %>
<% unless @keys_trusted.empty? -%>
trustedkey <%= @keys_trusted.join(' ') %>
<% end -%>
<% if @keys_requestkey != '' -%>
requestkey <%= @keys_requestkey %>
<% end -%>
<% if @keys_controlkey != '' -%>
controlkey <%= @keys_controlkey %>
<% end -%>

<% end -%>
<% [@fudge].flatten.each do |entry| -%>
fudge <%= entry %>
<% end -%>

<% unless @leapfile.nil? -%>
# Leapfile
leapfile <%= @leapfile %>
<% end -%>

```

Validating ERB templates

Before deploying a template, validate its syntax and render its output to make sure the template is producing the results you expect. Use the Ruby `erb` command to check Embedded Ruby (ERB) syntax.

ERB validation

To validate your ERB template, pipe the output from the `erb` command into `ruby`:

```
erb -P -x -T '-' example.erb | ruby -c
```

The `-P` switch ignores lines that start with `'%'`, the `-x` switch outputs the template's Ruby script, and `-T '-'` sets the trim mode to be consistent with Puppet's behavior. This output gets piped into Ruby's syntax checker (`-c`).

If you need to validate many templates quickly, you can implement this command as a shell function in your shell's login script, such as `.bashrc`, `.zshrc`, or `.profile`:

```
validate_erb() {
  erb -P -x -T '-' $1 | ruby -c
}
```

You can then run `validate_erb example.erb` to validate an ERB template.

Evaluating ERB templates

After you have an ERB template, you can pass it to a function that evaluates it and returns a final string. The actual template can be either a separate file or a string value.

Evaluating ERB templates that are in a template file

Put template files in the `templates` directory of a module. ERB files use the `.erb` extension.

To use a ERB template file, evaluate it with the `template` function. For example:

```
# template(<FILE REFERENCE>, [<ADDITIONAL FILES>, ...])
file { '/etc/ntp.conf':
  ensure => file,
  content => template('ntp/ntp.conf.erb'),
  # Loads /etc/puppetlabs/code/environments/production/modules/ntp/
  templates/ntp.conf.erb
}
```

The first argument to the function is the *file reference*: a string in the form `'<MODULE>/<FILE>'`, which loads `<FILE>` from `<MODULE>`'s templates directory. For example, the file reference `activemq/amq/activemq.xml.erb` loads the `<MODULES DIRECTORY>/activemq/templates/amq/activemq.xml.erb` file.

The template function can take any number of additional template files, and concatenate their outputs together to produce the final string.

Evaluating ERB template strings

If you have a string value that contains template content, you can evaluate it with the `inline_template` function.

In older versions of Puppet, inline templates were mostly used to get around limitations — tiny Ruby fragments were useful for transforming and manipulating data before Puppet had [iteration functions](#) like `map` or `puppetlabs/stdlib` functions like `chomp` and `keys`.

In modern versions of Puppet, inline templates are usable in some of the same situations template files are. Because the [heredoc](#) syntax makes it easy to write large and complicated strings in a manifest, you can use `inline_erb` to reduce the number of files needed for a simple module that manages a small config file.

For example:

```
$ntp_conf_template = @(END)
...template content goes here...
END

# inline_template(<TEMPLATE STRING>, [<ADDITIONAL STRINGS>, ...])
file { '/etc/ntp.conf':
  ensure => file,
  content => inline_template($ntp_conf_template),
}
```

The `inline_template` function can take any number of additional template strings, and concatenate their outputs together to produce the final value.

Advanced constructs

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

- [Iteration and loops](#) on page 570

Looping and iteration features help you write more succinct code, and use data more effectively.

- [Lambdas](#) on page 572

Lambdas are blocks of Puppet code passed to functions. When a function receives a lambda, it provides values for the lambda's parameters and evaluates its code. If you use other programming languages, think of lambdas as anonymous functions that are passed to other functions.

- [Resource default statements](#) on page 575

Resource default statements enable you to set default attribute values for a given resource type. Resource declarations within the area of effect that omits those attributes inherit the default values.

- [Resource collectors](#) on page 575

Resource collectors select a group of resources by searching the attributes of each resource in the catalog, even resources which haven't yet been declared at the time the collector is written. Collectors realize virtual resources, are used in chaining statements, and override resource attributes. Collectors have an irregular syntax that enables them to function as a statement and a value.

- [Virtual resources](#) on page 577

A virtual resource declaration specifies a desired state for a resource without enforcing that state. Puppet manages the resource by realizing it elsewhere in your manifests. This divides the work done by a normal resource declaration into two steps. Although virtual resources are declared one time, they can be realized any number of times, similar to a class.

- [Exported resources](#) on page 579

An exported resource declaration specifies a desired state for a resource, and publishes the resource for use by other nodes. It does not manage the resource on the target system. Any node, including the node that exports it, can collect the exported resource and manage its own copy of it.

- [Tags](#) on page 581

Tags are useful for collecting resources, analyzing reports, and restricting catalog runs. Resources, classes, and defined type instances can have multiple tags associated with them, and they receive some tags automatically.

- [Run stages](#) on page 583

Run stages are an additional way to order resources. Groups of classes run before or after everything else, without having to explicitly create relationships with other classes. The run stage feature has two parts: a `stage` resource type, and a `stage` metaparameter, which assigns a class to a named run stage.

Iteration and loops

Looping and iteration features help you write more succinct code, and use data more effectively.

Iteration functions

Iteration features are implemented as [functions](#) that accept blocks of code ([lambdas](#)). When a lambda requires extra information, pass it to a function that provides the information and to evaluate the code, possibly multiple times. This differs from some other languages where looping constructs are special keywords. In the Puppet code, they're functions.

Tip: Iteration functions take an [array](#) or a [hash](#) as their main argument, and iterate over its values.

These functions accept a block of code and run it in a specific way:

- `each` - Repeats a block of code a number of times, using a collection of values to provide different parameters each time.
- `slice` - Repeats a block of code a number of times, using groups of values from a collection as parameters.
- `filter` - Uses a block of code to transform a data structure by removing non-matching elements.
- `map` - Uses a block of code to transform every value in a data structure.
- `reduce` - Uses a block of code to create a new value, or data structure, by combining values from a provided data structure.
- `with` - Evaluates a block of code one time, isolating it in its own local scope. It doesn't iterate, but has a family resemblance to the iteration functions.

The [each](#), [filter](#), and [map](#) functions accept a lambda with either one or two parameters. Depending on the number of parameters, and the type of data structure you're iterating over, the values passed into a lambda vary:

Collection type	Single parameter	Two parameters
Array	<VALUE>	<INDEX>, <VALUE>
Hash	[<KEY>, <VALUE>] (two-element array)	<KEY>, <VALUE>

For example:

```
[ 'a', 'b', 'c' ].each | Integer $index, String $value | { notice("${index} = ${value}") }
```

Results in:

```
Notice: Scope(Class[main]): 0 = a
Notice: Scope(Class[main]): 1 = b
Notice: Scope(Class[main]): 2 = c
```

See the [slice](#) and [reduce](#) documentation for information on how these functions handle parameters differently.

Hashes preserve the order in which their keys and values were written. When iterating over a hash's members, the loops occur in the order that they are written. When interpolating a hash into a string, the resulting string is also constructed in the same order.

For information about the syntax of function calls, see the [functions documentation](#), or the [lambdas documentation](#) for information about the syntax of code blocks that you pass to functions.

Declaring resources

The focus of the Puppet language is declaring resources, so most people want to use iteration to declare many similar resources at the same time. In this example, there is an array of command names to be used in each symlink's path and target. The `each` function makes this succinct.

```
$binaries = ['factor', '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 a 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 else in your manifest file:

```
$binaries = ['factor', '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 $myparam = "default value"`). Each parameter consists of:
 - An optional [data type](#), which restricts the values it allows (defaults to Any).
 - A [variable](#) name to represent the parameter, including the \$ prefix.
 - An optional equals (=) sign and default value.
 - A closing pipe character (|).
 - Optionally, another comma and an extra arguments parameter (for example, `String *$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 positioned 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 must 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 must include a parameter value that changes with every call.

In this example, we use the `$binary` parameter in the title of the lambda's `file` resource:

```
file {["usr/bin/$binary":
  ensure => link,
  target => "/opt/puppetlabs/bin/$binary",
}]
```

When the `each` function is called, the array we pass has no repeated values to ensure unique `file` resources. However, if we are working with an array that came from less reliable external data, we could use [the `unique` function from `stdlib`](#) to protect against duplicates. This uniqueness requirement is [similar to defined types](#), which are also blocks of Puppet code that are evaluated multiple times.

Each time a lambda is called it produces the value of the last expression in the code block. The function that calls the lambda has access to this value, but not every function does anything with it. Some functions return it, some transform it, some ignore it, and some use it to do something else entirely.

For example:

- The `with` function calls its lambda one time and returns the resulting value.
- The `map` function calls its lambda multiple times and returns an array of every resulting value.
- The `each` function throws away its lambda's values and returns a copy of its main argument.

Every lambda creates its own [local scope](#) which is anonymous, and contains variables which can not be accessed by qualified names from any other scope. The parent scope of a lambda is the local scope in which that lambda is written. When a lambda is written inside a class definition, its code block accesses local variables from that class, as well as variables from that class's ancestor scopes, and from the top scope. Lambdas can contain other lambdas, which makes the outer lambda the parent scope of the inner one.

A lambda is a value with [the `Callable` data type](#), and functions using the modern function API (`Puppet::Functions`) use that data type to validate any lambda values it receives. However, the Puppet language doesn't provide any way to store or interact with `Callable` values except as lambdas provided to a function.

Resource default statements

Resource default statements enable you to set default attribute values for a given resource type. Resource declarations within the area of effect that omits those attributes inherit the default values.

Syntax

```
Exec {
  path      => '/usr/bin:/bin:/usr/sbin:/sbin',
  environment => 'RUBYLIB=/opt/puppetlabs/puppet/lib/ruby/site_ruby/2.1.0/',
  logoutput  => true,
  timeout    => 180,
}
```

The general form of resource defaults is:

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

You can specify defaults for any resource type in Puppet, including [defined types](#).

Behavior

Within the area of effect, each resource type that omits a given attribute uses that attribute's default value.

Important: Attributes set explicitly in a resource declaration override any default value.

Resource defaults are evaluation order dependent. Defaults are assigned to a created resource when a resource expression is evaluated; that is, when it is declared for inclusion in the catalog. Puppet uses the default values that are in effect for the type at evaluation.

Puppet uses dynamic scoping for resource defaults, even though it no longer uses dynamic variable lookup. This means that when you use a resource default statement in a class, it could affect any classes or defined types that class declares. Therefore, they should not be set outside of `site.pp`. [Use per-resource default attributes](#) when possible.

Resource defaults declared in the local scope override any defaults received from parent scopes. Overriding of resource defaults is per attribute, not per block of attributes. This means local and parent resource defaults that don't conflict with each other are merged together.

Resource collectors

Resource collectors select a group of resources by searching the attributes of each resource in the catalog, even resources which haven't yet been declared at the time the collector is written. Collectors realize virtual resources, are used in chaining statements, and override resource attributes. Collectors have an irregular syntax that enables them to function as a statement and a value.

Syntax

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

The general form of a resource collector is:

- A capitalized resource type name. This cannot be `Class`, and there is no way to collect classes.
- `<|` - An opening angle bracket (less-than sign) and pipe character.

- Optionally, a search expression.
- `| >` - A pipe character and closing angle bracket (greater-than sign)

Note: Exported resource collectors have a slightly different syntax; see below.

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

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

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

Use parentheses to improve readability, and to modify the priority and grouping of `and` and `or` operators. You can create complex expressions using four operators.

== (equality search)

This operator is non-symmetric:

- The left operand (attribute) is a string, and must be the name of a [resource attribute](#) or the word `title` (which searches on the resource's title). If the resource attribute name is a [reserved word](#) (for example, `site`, `unit`, or `application`), it must be in quotes.
- 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) is a string, and must be the name of a [resource attribute](#) or the word `title` (which searches on the resource's title). If the resource attribute name is a [reserved word](#) (for example, `site`, `unit`, or `application`), it must be in quotes.
- The right operand (search key) must be a [string](#), [boolean](#), [number](#), [resource reference](#), or `undef`. The behavior of arrays and hashes in the right operand is undefined in this version of Puppet.

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

Note: This operator always matches 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

Tip: Resource collector expressions does not produce a directly assignable value. However, the `query_resources` function of the `puppetdb_query` module returns an array of resources. For more information, see [PuppetDB query tools](#).

Behavior

A resource collector realizes any virtual resources matching its search expression. Empty search expressions match every resource of the specified resource type.

Note: A collector also collects and realizes any exported resources from the current node. When you use exported resources that you don't want realized, exclude them from the collector's search expression.

Collectors function as a value in two places:

- In a chaining statement, a collector acts as a proxy for every resource (virtual or not) that matches its search expression.
- When given a block of attributes and values, a collector sets and overrides those attributes for every resource (virtual or not) matching its search expression.

Note: Collectors used as values also realize any matching virtual resources. When you use virtualized resources, be careful when chaining collectors or using them for overrides.

Exported resource collectors

An exported resource collector uses a modified syntax that realizes exported resources and imports resources published by other nodes.

To use exported resource collectors, enable catalog storage and searching (`storeconfigs`). See Exported resources for more details. To enable exported resources, follow the installation instructions and Puppet configuration instructions in the [PuppetDB docs](#).

Like normal collectors, use exported resource collectors with attribute blocks and chaining statements.

Note: The search for exported resources also searches the catalog being compiled, to avoid having to perform an additional run before finding them in the store of exported resources.

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

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

The general form of an exported resource collector is:

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

Virtual resources

A virtual resource declaration specifies a desired state for a resource without enforcing that state. Puppet manages the resource by realizing it elsewhere in your manifests. This divides the work done by a normal resource declaration into two steps. Although virtual resources are declared one time, they can be realized any number of times, similar to a class.

Purpose

Virtual resources are useful for:

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

Because they both offer a safe way to add a resource to the catalog in multiple locations, virtual resources can be used in some of the same situations as [classes](#). The features that distinguish virtual resources are:

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

Syntax

Virtual resources are used in two steps: declaring and realizing. In this example, the `apache` class declares a virtual resource, and both the `wordpress` and `freight` classes realize it. The resource is managed on any node that has the `wordpress` or `freight` classes applied to it.

Declare: `modules/apache/manifests/init.pp`

```
@a2mod { 'rewrite':
  ensure => present,
} # note: The a2mod resource type is from the puppetlabs-apache module.
```

Realize: `modules/wordpress/manifests/init.pp`

```
realize A2mod['rewrite']
```

Realize again: `modules/freight/manifests/init.pp`

```
realize A2mod['rewrite']
```

To declare a virtual resource, prepend `@` (the “at” sign) to the resource type of a normal [resource declaration](#):

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

To realize one or more virtual resources by title, use the `realize` function, which accepts one or more [resource references](#):

```
realize(User['deploy'], User['zleslie'])
```

Note: The `realize` function can be used multiple times on the same virtual resource and the resource is managed only one time.

A [resource collector](#) realizes any virtual resources that match its search expression:

```
User <| tag == web |>
```

If multiple resource collectors match a given virtual resource, Puppet manages only that resource one time.

Note: A collector also collects and realizes any exported resources from the current node. If you use exported resources that you don’t want realized, take care to exclude them from the collector’s search expression. Also, a collector used in an [override block](#) or a [chaining statement](#) also realizes any matching virtual resources.

Behavior

A virtual resource declaration does not manage the state of a resource. Instead, it makes a virtual resource available to resource collectors and the `realize` function. When a resource is realized, Puppet manages its state.

Unrealized virtual resources are included in the [catalog](#), but are marked inactive.

Note: Virtual resources do not depend on evaluation order. You can realize a virtual resource before the resource has been declared.



CAUTION: The `realize` function causes a compilation failure when attempting to realize a virtual resource that has not been declared. Resource collectors fail silently when they do not match any resources.

When a virtual resource is contained in a class, it cannot be realized unless the class is declared at some point during the compilation. A common pattern is to declare a class full of virtual resources and then use a collector to choose the set of resources you need:

```
include virtual::users
User <| groups == admin or group == wheel |>
```

Note: You can declare virtual resources of defined resource types. This causes every resource contained in the defined resource to behave virtually — they are not managed unless their virtual container is realized.

Virtual resources are evaluated in the [run stage](#) in which they are declared, not the run stage in which they are realized.

Exported resources

An exported resource declaration specifies a desired state for a resource, and publishes the resource for use by other nodes. It does not manage the resource on the target system. Any node, including the node that exports it, can collect the exported resource and manage its own copy of it.

Purpose

Exported resources enable the Puppet compiler to share information among nodes by combining information from multiple nodes' catalogs. This helps manage things that rely on nodes knowing the states or activity of other nodes.

Note: Exported resources rely on the compiler accessing the information, and can not use information that's never sent to the compiler, such as the contents of arbitrary files on a node.

The common use cases are monitoring and backups. A class that manages a service like PostgreSQL, exports a `nagios_service` resource which describes how to monitor the service, including information such as its hostname and port. The Nagios server collects every `nagios_service` resource, and automatically starts monitoring the Postgres server.

Note: Exported resources require catalog storage and searching (`storeconfigs`) enabled on your primary Puppet server. Both the catalog storage and the searching, among other features, are provided by PuppetDB. To enable exported resources, see:

- [Install PuppetDB on a server at your site](#)
- [Connect your Puppet server 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    => ${::fqdn},
  check_command => 'check_nrpe_larg!check_zfs',
  service_description => "check_zfs${::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](#), which uses the [puppetlabs-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 primary server sends a copy of every compiled [catalog](#) to [PuppetDB](#). PuppetDB retains the recent catalog for every node and provides the server 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 primary server to send a search query to PuppetDB. PuppetDB responds with every exported resource that matches the [search expression](#), and the server 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 primary server must have compiled a given node's catalog at least one time before its resources become available. When the primary server submits a catalog to PuppetDB, it is added to a queue and stored as soon as possible. Depending on the PuppetDB server's workload, there might be a delay between a node's catalog being compiled and its resources becoming available.

Normally, exported resource types include their default attribute values. However, defined types are evaluated for the catalog after the resource is collected, so their default values are not exported. To make sure a defined type's values are exported, set them explicitly.

To remove stale exported resources, expire or deactivate the node that exported them. This ensures that any resources exported by that node stop appearing in the catalogs served to the remaining agent nodes. For details, see the documentation for [deactivating or expiring nodes](#).



CAUTION: Each exported resource must be globally unique across every single node. If two nodes export resources with the same [title](#) or same [name/namevar](#), the compilation fails when you attempt to collect both. Some pre-1.0 versions of PuppetDB do not fail in this case. To ensure uniqueness, every resource you export must include a substring unique to the node exporting it into its title and name/namevar. The most expedient way is to use the `hostname` or `fqdn` facts.

Restriction: Exported resource collectors do not collect normal or virtual resources. They cannot retrieve non-exported resources from other nodes' catalogs.

The following example shows Puppet native types for managing Nagios configuration files. These types become powerful when you export and collect them.

For example, to create a class for Apache that adds a service definition on your Nagios host, automatically monitoring the web server:

`/etc/puppetlabs/puppet/modules/nagios/manifests/target/apache.pp:`

```
class nagios::target::apache {
  @@nagios_host { $::fqdn:
    ensure => present,
    alias  => $::hostname,
    address => $::ipaddress,
    use    => 'generic-host',
  }
  @@nagios_service { "check_ping_${::hostname}":
    check_command => 'check_ping!100.0,20%!500.0,60%',
    use           => 'generic-service',
    host_name     => $::fqdn,
    notification_period => '24x7',
    service_description => "${::hostname}_check_ping"
  }
}
```

`/etc/puppetlabs/puppet/modules/nagios/manifests/monitor.pp:`

```
class nagios::monitor {
  package { [ 'nagios', 'nagios-plugins' ]: ensure => installed, }
  service { 'nagios':
    ensure      => running,
    enable      => true,
    #subscribe => File[$nagios_cfgdir],
    require     => Package['nagios'],
  }
}
```

Collect resources and populate `/etc/nagios/nagios_*.cfg`:

```
Nagios_host <<||>>
Nagios_service <<||>>
}
```

Tags

Tags are useful for collecting resources, analyzing reports, and restricting catalog runs. Resources, classes, and defined type instances can have multiple tags associated with them, and they receive some tags automatically.

Tag names

For information about the characters allowed in tag names, see [reserved words and acceptable names](#).

Assigning tags to resources

Every resource automatically receives the following tags:

- Its resource type.
- The full name of the [class](#) or [defined type](#) in which the resource was declared.
- Every [namespace segment](#) of the resource's class or defined type.

For example, a file resource in class `apache::ssl` is automatically assigned the tags `file`, `apache::ssl`, `apache`, and `ssl`. Do not manually assign tags with names that are the same as these automatically assigned tags.

Tip: Class tags are useful when setting up the [tagmail](#) module or testing refactored manifests.

Similar to [relationships](#) and most metaparameters, tags are passed along by [containment](#). This means a resource receives all of the tags from the class and/or defined type that contains it. In the case of nested containment (a class

that declares a defined resource, or a defined type that declares other defined resources), a resource receives tags from all of its containers.

The `tag` metaparameter accepts a single tag or an array, and these are added to the tags the resource already has. A tag can also be used with normal resources, [defined resources](#), and classes (when using the resource-like declaration syntax).

Because [containment](#) applies to tags, the example below assigns the `us_mirror1` and `us_mirror2` tags to every resource contained by `Apache::Vhost['docs.puppetlabs.com']`.

To add multiple tags, use [the tag metaparameter](#) in a resource declaration:

```
apache::vhost {'docs.puppetlabs.com':
  port => 80,
  tag   => ['us_mirror1', 'us_mirror2'],
}
```

To assign tags to the surrounding container and all of the resources it contains, use [the tag function](#) inside a class definition or defined type. The example below assigns the `us_mirror1` and `us_mirror2` tags to all of the defined resources being declared in the class `role::public_web`, as well as to all of the resources each of them contains.

```
class role::public_web {
  tag 'us_mirror1', 'us_mirror2'

  apache::vhost {'docs.puppetlabs.com':
    port => 80,
  }
  ssh::allowgroup {'www-data': }
  @nagios::website {'docs.puppetlabs.com': }
}
```

Using tags

Tip: Tags are useful when used as an attribute in the [search expression](#) of a [resource collector](#) for realizing [virtual](#) and [exported](#) resources.

Puppet agent and Puppet apply use [the tags setting](#) to apply a subset of the node's [catalog](#). This is useful when refactoring modules, and enables you to apply a single class on a test node.

The `tags` setting can be set in `puppet.conf` to restrict the catalog, or on the command line to temporarily restrict it. The value of the `tags` setting must 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 `hier` while using [resource-like](#) class declarations.
- You cannot subscribe to or notify resources across a stage boundary.
- Classes that [contain](#) other classes, with either the `contain` function or the anchor pattern, can sometimes behave badly if declared with a run stage. If the contained class is declared only by its container, it works fine, but if it's declared anywhere outside its container, it often creates a dependency cycle that prevents the involved classes being applied.



CAUTION: Due to these limitations, use stages with the simplest of classes, and only when absolutely necessary. A valid use case is mass dependencies like package repositories.

Details of complex behaviors

Within Puppet language there are complex behavior patterns regarding classes, defined types, and specific areas of code called scopes.

- [Containment](#) on page 584

Containment is what controls the order in which the various parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types, determining what has to happen before other things can happen.

- [Scope](#) on page 586

A scope is a specific area of code that is partially isolated from other areas of code.

- [Namespaces and autoloading](#) on page 590

Class and defined type names can be broken up into segments called namespaces which enable the autoloader to find the class or defined type in your modules.

Containment

Containment is what controls the order in which the various parts of your Puppet code are executed. Containment is the relationship that resources have to classes and defined types, determining what has to happen before other things can happen.

[Classes](#) and [defined type](#) instances *contain* the resources they declare. Contained resources are not applied before the container begins, and they finish before the container finishes.

This means that if any resource or class forms a [relationship](#) with the container, it forms the same relationship with every resource inside the container.

Consider this example:

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

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

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

Containment allows you to [notify and subscribe to](#) classes and defined resource types as though they were a single resource.

Containment of resources

Resources of both native and defined resource types are automatically contained by the class or defined type in which they are declared.

Containment of classes

Unlike with resources, Puppet does not automatically contain classes when they are declared inside another class (by using the `include` function or resource-like declaration). But in certain situations, having classes contain other classes can be useful, especially in larger modules where you want to improve code readability by moving chunks of implementation into separate files.

You can declare a class in any number of places in the code, allowing classes to announce their needs without worrying about whether other code also needs the same classes at the same time. Puppet includes the declared class

only one time, regardless of how many times it's declared (that is, the `include` function is *idempotent*). Usually, this is fine, and code shouldn't attempt to strictly contain the class. However, there are ways to explicitly set more strict containment relationships of contained classes when it is called for.

When you're deciding whether to set up explicit containment relationships for declared classes, follow these guidelines:

- **include:** When you need to declare a class and nothing in it is required for the enforcement of the current class you're working on, use the `include` function. It ensures that the named class is included. It sets no ordering relationships. Use `include` as your default choice for declaring classes. Use the other functions only if they meet specific criteria.
- **require:** When resources from another class should be enforced before the current class you're working on can be enforced properly, use the `require` function to declare classes. It ensures that the named class is included. It sets relationships so that everything in the named class is enforced before the current class.
- **contain:** When you are writing a class in which users should be able to set relationships, use the `contain` function to declare classes. It ensures that the named class is included. It sets relationships so that any relationships specified on the current class also apply to the class you're containing.

The require function

The `require` function is useful when the class you're writing needs another class to be successfully enforced before it can be enforced properly.

For example, suppose you're writing classes to install two apps, both of which are distributed by the Chocolatey package manager, for which you've created a class called `chocolatey`. Both classes require that Chocolatey be properly managed before they can each proceed. Instead of using `include`, which won't ensure Chocolatey's resources are managed before it installs each app, use `require`.

```
class myapp::install {
  # works just like include, but also creates a relationship
  # Class['chocolatey'] -> Class['myapp::install']
  require chocolatey
  package { 'myapp':
    ensure => present,
  }
}

class my_other_app::install {
  require chocolatey
  package { 'my_other_app':
    ensure => present,
  }
}
```

The contain function

Use the `contain` function to declare that a class is contained. This is useful for when you're writing a class in which other users should be able to express relationships. Any classes contained in your class will have containment relationships with any other classes that declare your class. The `contain` function uses include-like behavior, containing a class within a surrounding class.

For example, suppose you have three classes that an app package (`myapp::install`), creating its configuration file (`myapp::config`), and managing its service (`myapp::service`). Using the `contain` function explicitly tells Puppet that the internal classes should be contained within the class that declares them. The `contain` function works like `include`, but also adds class relationships that ensure that relationships made on the parent class also propagate inside, just like they do with resources.

```
class myapp {
  # Using the contain function ensures that relationships on myapp also
  # apply to these classes
```

```

contain myapp::install
contain myapp::config
contain myapp::service

Class['myapp::install'] -> Class['myapp::config'] ~>
Class['myapp::service']
}

```

Although it may be tempting to use `contain` everywhere, it's better to use `include` unless there's an explicit reason why it won't work.

Scope

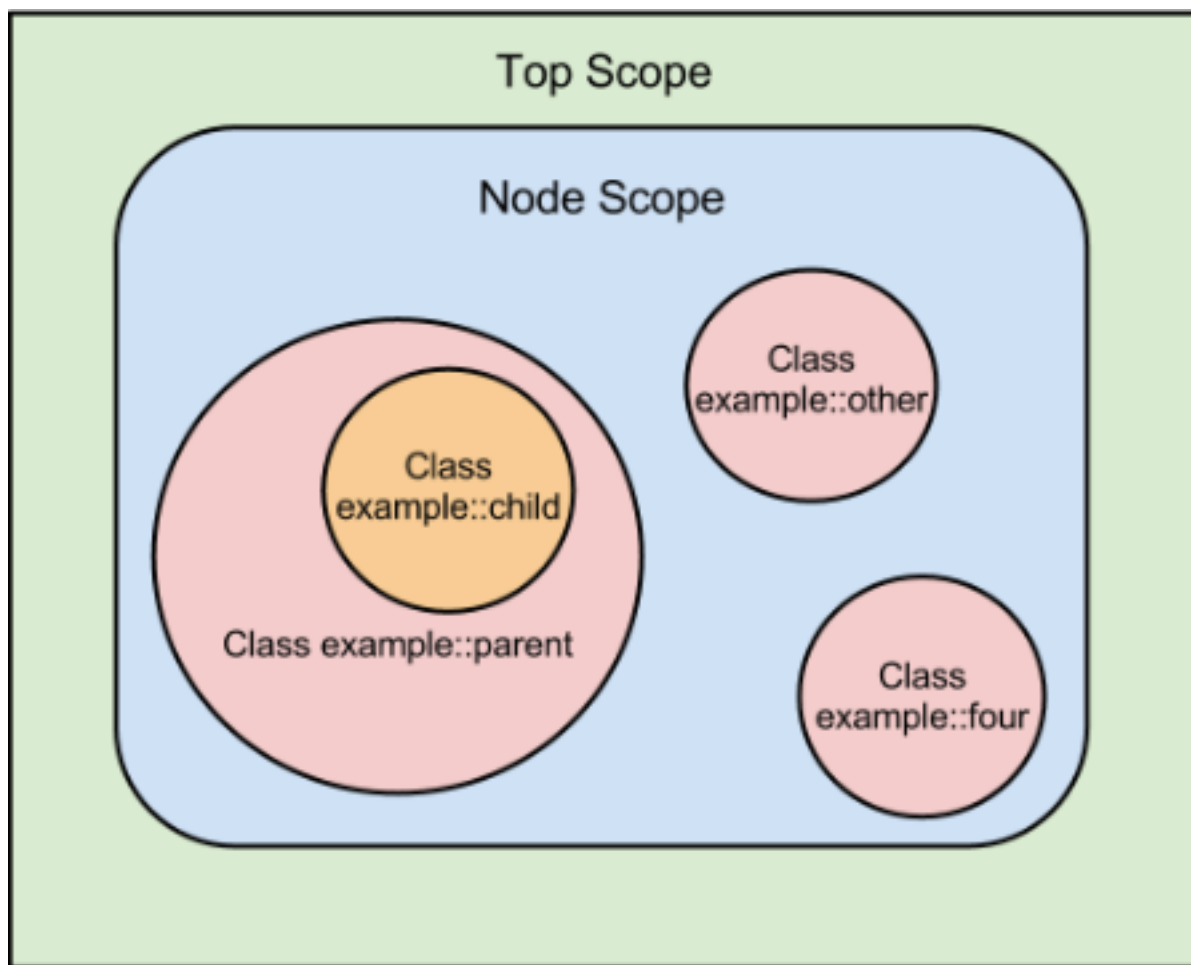
A scope is a specific area of code that is partially isolated from other areas of code.

Scopes limit the reach of:

- [Variables](#).
- [Resource defaults](#).

Scopes do not limit the reach of:

- [Resource titles](#), which are all global.
- [Resource references](#), which can refer to a resource declared in any scope.



A particular scope has access to its own contents, and also receives additional contents from its parent scope, node scope, and top scope. The rules for how Puppet determines a local scope's parent are described in [scope lookup rules](#).

In the diagram above:

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

Top scope

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

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

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

include example
```

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

Node scope

Code inside a [node definition](#) exists at node scope. Only one node scope can exist at a time because only one node definition can match a given node.

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

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

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

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

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

Local scopes

Code inside a [class definition](#), [defined type](#), or [lambda](#) exists in a local scope.

Variables and defaults declared in a local scope are only available in that scope and its children. There are two different sets of rules for when scopes are considered related. For more information, see [scope lookup rules](#).

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

```
# /etc/puppetlabs/code/modules/scope_example/manifests/init.pp
```

```

class scope_example {
  $variable = "Hi!"
  notify {"Message from here: $variable":}
  notify {"Node scope: $node_variable Top scope: $top_variable":}
}

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

```

```

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

```

Overriding received values

Variables and defaults declared at node scope can override those received from top scope. Those declared at local scope can override those received from node and top scope, as well as any parent scopes. If multiple variables with the same name are available, Puppet uses the most local one.

```

# /etc/puppetlabs/code/modules/scope_example/manifests/init.pp
class scope_example {
  $variable = "Hi, I'm local!"
  notify {"Message from here: $variable":}
}

# /etc/puppetlabs/code/environments/production/manifests/site.pp
$variable = "Hi, I'm top!"

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

```

```

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

```

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

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

```

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

  file {'/tmp/example':}
}

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

```

```
}
include scope_example
```

Scope of external node classifier data

Variables provided by an [ENC](#) are set at the top scope. However, all of the classes assigned by an ENC are declared at the node scope.

This results the most expected behavior: variables from an ENC are available everywhere, and classes can use node-specific variables.

Note: This means compilation fails if a site manifest tries to set a variable that was already set at top scope by an ENC.

Node scope only exists if there is at least one node definition in the main manifest. If no node definitions exist, then ENC classes get declared at top scope.

Named scopes and anonymous scopes

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

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

Accessing out-of-scope variables

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

Qualified variable names are formatted using the double-colon [namespace](#) separator between segments:

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

In the following example, the variable `$local_copy` is set to the value of the `$confdir` variable from the `apache::params` class:

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

Note:

A class must be [declared](#) to access its variables; just having the class available in your modules is insufficient.

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

Because the top scope's name is the empty string, `$::my_variable` refers to the top-scope value of `$my_variable`, even if `$my_variable` has a different value in local scope.

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

Scope lookup rules

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

There are two sets of scope lookup rules: static scope and dynamic scope. Puppet uses:

- Static scope for [variables](#).
- Dynamic scope for [resource defaults](#).

Static scope

In static scope, which Puppet uses for looking up variables, parent scopes are assigned in the following ways:

- Classes can receive parent scopes by [class inheritance](#), using the `inherits` keyword. A derived class receives the contents of its base class in addition to the contents of node and top scope.
- A [lambda's](#) parent scope is the local scope in which the lambda is written. It can access variables in that scope by their short names.

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

Note: Static scope has the following characteristics:

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

Dynamic scope

In dynamic scope, which Puppet uses for looking up resource defaults, parent scopes are assigned by both inheritance and declaration, with preference given to inheritance. The full list of rules is:

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

Note: Dynamic scope has the following characteristics:

- A scope's parent cannot be identified by looking at the definition of a class — you must examine every place where the class or resource might have been declared.
- In some cases, you can only determine a scope's contents by executing the code.
- Because classes can be declared multiple times with the `include` function, the contents of a given scope are evaluation-order dependent.

Namespaces and autoloading

Class and defined type names can be broken up into segments called namespaces which enable the autoloader to find the class or defined type in your modules.

Syntax

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

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

Autoloader behavior

When a class or defined resource is declared, Puppet uses its full name to find the class or defined type in your modules. Every class and defined type must be in its own file in the module's `manifests` directory, and each file must have the `.pp` file extension.

Names map to file locations as follows:

- The first segment in a name, excluding the empty top namespace, identifies the [module](#). If this is the only segment, the file name is `init.pp`.
- The last segment identifies the file name, minus the `.pp` extension.
- Any segments between the first and last are subdirectories under the `manifests` directory.

As a result, every class or defined type name maps directly to a file path within Puppet's [modulepath](#):

Name	File path
<code>apache</code>	<code><MODULE DIRECTORY>/apache/manifests/init.pp</code>
<code>apache::mod</code>	<code><MODULE DIRECTORY>/apache/manifests/mod.pp</code>
<code>apache::mod::passenger</code>	<code><MODULE DIRECTORY>/apache/manifests/mod/passenger.pp</code>

Note: The `init.pp` file always contains a class or defined type with the same name as the module, and any other `.pp` file contains a class or defined type with at least two namespace segments. For example, `apache.pp` would contain a class named `apache::apache`. This means you can't name a class `<MODULE NAME>::init`.

Nested definitions and missing files

If a class or defined type is defined inside another class or defined type definition, its name goes under the outer definition's namespace.

This causes its real name to be something other than the name it was defined with. For example, in the following code, the interior class's real name is `first::second`:

```
class first {
  class second {
    ...
  }
}
```

However, searching your code for that real name returns nothing. Also, it causes class `first::second` to be defined in the wrong file. **Avoid structuring your code like this.**

If the manifest file that corresponds to a name doesn't exist, Puppet continues to look for the requested class or defined type. It does this by removing the final segment of the name and trying to load the corresponding file, continuing to fall back until it reaches the module's `init.pp` file.

Puppet loads the first file it finds like this, and raises an error if that file doesn't contain the requested class or defined type.

This behavior allows you to put a class or defined type in the wrong file and still have it work. But structuring things this way is not recommended.

Modules

Modules manage a specific technology in your infrastructure and serve as the basic building blocks of Puppet desired state management.

Helpful modules docs links	Other useful places
Understanding Puppet modules Modules overview Plug-ins in modules Modules cheatsheet	Modules on the Puppet Forge The Forge module repository Puppet approved modules Puppet supported modules module quality scoring
Managing modules Installing modules Upgrading modules Uninstalling modules puppet-module command reference	Modules in Code Manager Managing modules with the Puppetfile
Writing modules Beginner's guide to modules Module metadata Publishing modules	Puppet language reference Classes Defined types Puppet tasks and plans
Documenting modules Writing module documentation Puppet Strings Puppet Strings style guide	Developing and testing modules Puppet Development Kit (PDK)
Publishing modules On the Forge	Developing types and providers Puppet Resource API
Contributing to modules Contributing to Puppet modules Reviewing community pull requests	Community resources Puppet Community Slack puppet-users email group

Modules overview

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. 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.

The following video gives you an overview of modules:

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, and not all directories are required. Use the following directory 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 primary Puppet server.

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 primary Puppet server and the Puppet agent, and they are synced to all agent nodes in the environment on each Puppet run.

factor/

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.

plans/

Contains Puppet task plans, which are sets of tasks that can be combined with other logic. Plans are written in the Puppet language.

readmes/

The module's README localized into languages other than English.

spec/

Contains spec tests for any plug-ins in the lib directory.

tasks/

Contains Puppet tasks, which can be written in any programming language that can be read by the target node.

templates/

Contains templates, which the module's manifests can use to generate content or variable values.

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 must match the expression: `[a-z][a-z0-9_]*`. In other words, they can contain only lowercase letters, numbers, and underscores, and begin with a lowercase letter.

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 separator.
- 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 returns the content of the requested file from the module's `files/` directory. Puppet URLs work for both `puppet agent` and `puppet apply`; in either case they retrieve the file from a module.

To learn more about the `file` function, see the [function reference](#).

Templates in modules

You can use ERB or EPP templates in your module to manage the content of configuration files. Templates combine code, data, and literal text to produce a string output, which can be used as the content attribute of a `file` resource or as a variable value. Templates are contained in the module's `templates/` directory.

For ERB templates, which use Ruby, use the `template` function. For EPP templates, which use the Puppet language, use the `epp` function. See the page about [templates](#) for detailed information.

The `template` and `epp` functions look up templates identified by module and template name, passed as a string in parentheses: `function('module_name/template_name.extension')`. For example:

```
template('my_module/component.erb')
```

```
epp('my_module/component.epp')
```

Writing modules

Every Puppet user can expect to write at least some of their own modules. You must give your modules a specific directory structure and include correctly formatted metadata. Puppet Development Kit (PDK) provides tools for writing, validating, and testing modules.

PDK creates a complete module structure, class, defined type, and task templates, and configures a module testing framework. To test your modules, use PDK commands to run unit tests and to validate your module's metadata,

syntax, and style. You can download and install PDK on any development machine; no Puppet installation is required. See the PDK [documentation](#) to get started.

For help getting started writing modules, see our beginner's guide to writing modules. For details on best practices and code style, see the Puppet Language style guide.

Related information

[Beginner's guide to writing modules](#) on page 607

Create great Puppet modules by following best practices and guidelines.

[Documenting modules](#) on page 618

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 355

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 primary server 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 Factor custom facts documentation.	Agents only.	<code>lib/facter</code>
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 Factor custom facts documentation.	Agents only.	<code>facts.d</code>
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.	<code>functions</code>
Ruby functions	Functions written in Ruby to return calculated values. Modern Ruby functions are written for the <code>Puppet::Functions</code> API.	Puppet Server only.	<code>lib/puppet/functions</code>
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.	<code>lib/puppet/type</code>
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.	<code>lib/puppet/provider</code>
Augeas lenses	Augeas provides a way to modify config files. To learn more about using Augeas with Puppet, see the Forge for Augeas tips and tricks .	Agents only.	<code>lib/augeas/lenses</code>

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

[Modules overview](#) on page 592

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 596

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 [Puppet configuration 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 puppetlabs-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
```

Installing from another module repository

You can install modules from other repositories that mimic the Forge interface. You can change the module repository for one installation, or you can change your default repository.

To change the module repository for a single module installation, specify the base URL of the repository on the command line with the `--module_repository` option. For example:

```
puppet module install --module_repository http://dev-forge.example.com
puppetlabs-apache
```

To change the default module repository, edit the `module_repository` setting in the `puppet.conf` to the base URL of the repository you want to use. The default value for the `module_repository` is the Forge URL, `https://forgeapi.puppetlabs.com`. See the `module_repository` setting in the [puppet.conf configuration](#) documentation.

Installing from a release tarball

To install a module from a release tarball, specify the path to the tarball instead of the module name.

If you cannot connect to the Forge, or you are installing modules that have not yet been published to the Forge, use the `--ignore-dependencies` option and manually install any dependencies. For example:

```
puppet module install ~/puppetlabs-apache-0.10.0.tar.gz --ignore-
dependencies
```

Installing and upgrading Puppet Enterprise-only modules

Some Puppet modules are available only to PE users. Generally, you manage these modules in the same way you would manage other modules. You can use these modules with licensed PE nodes, a PE 10-node trial license, or with Bolt for a limited evaluation period. See your module's license for complete details.

Install modules on nodes without internet

To manually install a module on a node with no internet, download the module on a connected machine, and then move a module package to the unconnected node. If the module is a PE-only module, the download machine must have a valid PE license.

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 after it was installed. You can forcibly uninstall dependencies or changed modules with command line options.

For example, to uninstall a module that other modules depend on, run:

```
puppet module uninstall --force
```

See the `puppet module` command reference for a complete list of options.

puppet module command reference

The `puppet module` command manages modules with several actions and options.

puppet module actions

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.	<code>puppet module build modules/apache</code>
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.	<code>puppet module changes /etc/code/modules/stdlib</code>
install	Installs a module.	The full name <code><username-module_name></code> of the module to uninstall.	<code>puppet module install puppetlabs-apache</code>
list	Lists the modules installed in the modulepath specified in the <code>[main]</code> block in the <code>puppet.conf</code> file.	None.	<code>puppet module list</code>
search	Searches the Forge for modules matching search values.	A single search term.	<code>puppet module search apache</code>
uninstall	Uninstalls a module.	The full name <code><username-module_name></code> of the module to uninstall.	<code>puppet module uninstall puppetlabs-apache</code>
upgrade	Upgrades a module to the most recent release or to the specified version. Does not upgrade dependencies.	The full name <code><username-module_name></code> of the module to upgrade.	<code>puppet module upgrade puppetlabs-apache --version 0.0.3</code>

puppet module install action

Installs a module from the Forge or another specified release archive.

Usage:

```
puppet module install [--debug] [--environment] [--force | -f] [--ignore-
dependencies]
  [--module_repository <REPOSITORY_URL>] [--strict-semver]
  [--target-dir <DIRECTORY/PATH> | -i <DIRECTORY/PATH>] [--version <x.x.x> |
-v <x.x.x>]
  <full_module_name>
```

For example:

```
puppet module install --environment testing --ignore-dependencies
--version 1.0.0-pre1 --strict-semver false puppetlabs-apache
```

Option	Description	Value	Default
--debug, -d	Displays additional information about what the puppet module command is doing.	None.	If not specified, additional information is not displayed.
--environment	Installs the module into the specified environment.	An environment name.	By default, installs the module into the default environment specified in the puppet.conf file.
--force, -f	Installs the module regardless of dependency tree, checksum changes, or whether the module is already installed. By default, installs the module in the default modulepath, even if the module is already installed in another directory. Does not install dependencies.	None.	If not specified, puppet module install exits and returns information if it encounters installation errors or conflicts.
--ignore-dependencies	Does not install any modules required by this module.	None.	If not specified, the puppet module install action installs the module and its dependencies.
--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

Option	Description	Value	Default
<code>--version, -v</code>	Specifies the module version to install.	A semantic version number, such as 1.2.1 or a string specifying a requirement, such as <code>">=1.0.3"</code> .	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
<code>--modulepath</code>	Specifies another modulepath to scan for modules.	A valid directory path.	By default, scans the default modulepath from the <code>[main]</code> block in the <code>puppet.conf</code> file.
<code>--strict-semver</code>	Whether to exclude pre-release versions. A value of <code>false</code> allows uninstallation of pre-release versions.	<code>true, false</code>	Defaults to <code>true</code> , excluding pre-release versions.
<code>--tree</code>	Displays the module list as a tree showing dependencies.	None.	By default, <code>puppet module list</code> 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
<code>--force, -f</code>	Uninstalls the module regardless of dependency tree or checksum changes.	None.	By default, <code>puppet module uninstall</code> exits and returns an error if it encounters changes, namespace errors, or dependencies.
<code>--ignore-changes</code>	Does not use the checksum and uninstalls regardless of modified files.	None.	By default, if the <code>puppet module uninstall</code> action finds modified files in the module, it exits and returns an error.
<code>--strict-semver</code>	Whether to exclude pre-release versions. A value of <code>false</code> allows uninstallation of pre-release versions.	<code>true, false</code>	Defaults to <code>true</code> , excluding pre-release versions.
<code>--version, -v</code>	Specifies the module version to uninstall.	A semantic version number, such as <code>1.2.1</code> or a string specifying a requirement, such as <code>">=1.0.3"..</code>	By default, <code>puppet module uninstall</code> 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
<code>--force, -f</code>	Upgrades the module regardless of dependency tree or checksum changes.	None.	By default, <code>puppet module upgrade</code> exits and returns an error if it encounters changes, namespace errors, or dependencies.
<code>--ignore-changes</code>	Does not use the checksum and upgrades regardless of modified files.	None.	By default, if the <code>puppet module upgrade</code> action finds modified files in the module, it exits and returns an error.

Option	Description	Value	Default
<code>--ignore-dependencies</code>	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.
<code>--strict-semver</code>	Whether version ranges must exclude pre-release versions.	true, false	Defaults to true, excluding pre-release versions.
<code>--version, -v</code>	Specifies the module version to uninstall.	A semantic version number, such as 1.2.1 or a string specifying a requirement, such as <code>">=1.0.3"</code> .	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 specifying

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 is named `init.pp`. This class is the module's main interface point with Puppet. If possible, make the main class the only parameterized class in your module. Limiting the parameterized classes to only the main class means that you only have to include a single class to control usage of the entire module. This class provides 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 contains 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

Place the resources related to configuring the installed software 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

Put the remaining service resources, and anything else related to the running state of the software 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": "puppetlabs",
  "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" }
  ],
  "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"
    }
  ]
}

```

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.
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 `["mysql", "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 must 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 must match an identifier provided by SPDX. For a complete list, see the SPDX license list .	"Apache-2.0"
"summary"	Required.	A one-line description of your module.	"Standard library of resources for Puppet modules."

Key	Required?	Value	Example
"source"	Required.	The source repository for your module.	"https://github.com/puppetlabs/puppetlabs-stdlib"
"dependencies"	Required.	An array of other modules that your module depends on to function. If the module has no dependencies, pass an empty array. See the related topic about specifying dependencies for more details.	<pre>"dependencies": [{ "name": "puppetlabs/stdlib", "version_requirement": ">= 4.13.1 < 6.0.0" }],</pre>
"requirements"	Optional.	A list of external requirements for your module, given as an array of hashes.	<pre>"requirements": [{ "name": "puppet", "version_requirement": ">= 4.7.0 < 6.0.0" }],</pre>
"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.	<pre>{ "operatingsystem": "RedHat", "operatingsystemrelease": ["5", "6", "7"] }</pre>
"tags"	Optional.	An array of four to six key words to help people find your module.	["mysql", "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.

Use clear and consistent language in your Module documentation. It should be easy to read both on the web and in the 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 must do. For example, "Secure your dog door before installing the cat module."
- Use the active voice whenever possible. For example, "Install the cat and bird modules on separate instances" rather than "The cat and bird modules should be installed on separate instances."
- Use the present tense, almost always. Events that regularly occur should be present tense: "This parameter sets your cat to 'purebred'. The purebred cat alerts you for breakfast at 6 a.m." Use future tense only when you are specifically referring to something that takes place at a time in the future, such as "The `tail` parameter is deprecated and will be removed in a future version. Use `manx` instead."
- Avoid subjective words. For example, don't write "It's quick and easy to teach an old cat new tricks." Subjective words like "quick" and "easy" can frustrate and even alienate a reader who finds teaching a cat difficult or time-consuming.
- Lists, whether ordered or unordered, make things clearer for the reader. When you're writing about steps that 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 622

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 629

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 tells 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. 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](#).

Use the following sections in your README:

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.

The short description helps the user decide if your module is what they want. What are the most common use cases for your module? Does your module just install software? Does it install and configure it? Give your user information about what to expect from the module.

```
## Module description

The `cat` module installs, configures, and maintains your cat in both
apartment and residential house settings.

The cat module automates the installation of a cat to your apartment or
house, and then provides options for configuring the cat to fit your
environment's needs. After it's installed and configured, the cat module
```

```
automates maintenance of your cat through a series of resource types and providers.
```

Setup section

In the setup section, detail how your user can successfully get your module functioning. Include requirements, steps to get started, and any other information users might need to know before they start using your module.

Module installation instructions are covered both on the module's Forge page and in the Puppet docs, so don't reiterate them here. In this section, include the following subsections, as applicable:

What <modulename> affects

Include this section only if:

- The module alters, overwrites, or otherwise touches files, packages, services, or operations other than the named software; OR
- The module's general performance can overwrite, purge, or otherwise remove entries, files, or directories in a user's environment. For example:

```
## Setup

### What cat affects

* Your dog door might be overwritten if not secured before
  installation.
```

Setup requirements

Include this section only if the module requires additional software or some tweak to a user's environment. For instance, the `puppetlabs-firewall` module uses Ruby-based providers which required `pluginsync` to be enabled.

Beginning with <modulename>

Always include this section to explain the minimum steps required to get the module up and running in a user's environment. You can use basic proof of concept use cases here; it doesn't have to be something you would run in production. For simple modules, "Declare the main `::cat` class" is enough.

Usage section

Include examples for common use cases in the usage section. Provide usage information and code examples to show your users how to use your module to solve problems.

If there are many use cases for your module, include three to five examples of the most important or common tasks a user can accomplish. The usage section is a good place to include more complex examples that involve different types, classes, and functions working together. For example, the usage section for the `puppetlabs-apache` module includes an example for setting up a virtual host with SSL, which involves several classes.

```
## Usage

You can manage all interaction with your cat through the main `cat`
class. With the default options, the module installs a basic cat with no
optimizations.

### I just want cat, what's the minimum I need?
```
include '::cat'
```

### I want to configure my lasers
```

Use the following to configure your lasers for a random-pattern, 20-minute playtime at 3 a.m. local time.

```

...
class { 'cat':
  laser => {
    pattern    => 'random',
    duration   => '20',
    start_time => '0300',
  }
}
...

```

Limitations section

In the limitations section, list any incompatibilities, known issues, or other warnings.

```
## Limitations
```

```
This module cannot be used with the smallchild module.
```

Development section

In the development section, tell other users the ground rules for contributing to your project and explain how they should submit their work.

Creating reference documentation

List reference information --- a complete list of classes, defined types, functions, resource types and providers, tasks, and plans --- in a separate `REFERENCE.md` file in the root directory of your module.

Use Puppet Strings to generate this documentation based on your comments and module code. If you aren't yet using Strings to generate documentation, you can manually create a `REFERENCE.md` file.

Tip: Previously, we recommended that module authors include reference information in the README itself. However, the reference section often became quite long and difficult to maintain. Moving reference information to a separate file keeps the README more readable, and using Strings to generate this file makes it easier to maintain.

The Forge displays information from a module's `REFERENCE.md` file in a reference tab on the module's detail page, so the information remains easily accessible to users. To create a `REFERENCE.md` file for your module, add Strings comments to the code for each of your classes, defined types, functions, task plans, and resource types and providers, and then run Strings to generate documentation in Markdown. You can create a `REFERENCE.md` file manually, but remember that if you then generate a `REFERENCE.md` with Strings, it overwrites any existing `REFERENCE.md` file.

For details on adding comments to your code, see the [Strings style guide](#). For instructions on how to install and use Strings, see the topics about Puppet Strings.

Manually writing reference documentation

If you aren't using Strings yet to generate your reference documentation, you can manually create a `REFERENCE.md` file listing each of your classes, defined types, resource types and providers, functions, and facts, along with any parameters.

To manually document reference information, start your reference document with a small table of contents that first lists the classes, defined types, and resource types of your module. If your module contains both public and private classes or defined types, list the public and the private separately. Include a brief description of what these items do in your module.

```
## Reference
```

```
### Classes
```

```
#### Public classes

*[\pet::cat\](#petcat): Installs and configures a cat in your environment.

#### Private classes

*[\pet::cat::install\]: Handles the cat packages.
*[\pet::cat::configure\]: Handles the configuration file.
```

After this table of contents, list the parameters, providers, or features for each element (class, defined type, function, and so on) of your module. Be sure to include valid or acceptable values and any defaults that apply. Each element in this list must 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 618

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 629

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/apache/lib/**/*.rb' 'modules/apache/manifests/**/*.pp' 'modules/apache/functions/**/*.pp'
```

Strings outputs HTML to the `doc/` directory in the module. To view the generated HTML documentation for a module, open the `index.html` file in the module's `doc/` folder. To view HTML documentation for all of your local modules, run `puppet strings server` from any directory. This command serves documentation for all modules in the module path at `http://localhost:8808`. To learn more about the modulepath, see the [modulepath](#) documentation.

Generate and view documentation in Markdown

To generate reference documentation in Markdown, specify the `markdown` format when you run Puppet Strings.

The reference documentation includes descriptions, usage details, and parameter information for classes, defined types, functions, tasks, plans, and resource types and providers.

Strings generates Markdown output as a `REFERENCE.md` file in the main module directory, but you can specify a different filename or location with command line options.

1. Change directory into the module: `cd /modules/<MODULE_NAME>`
2. Run the command: `puppet strings generate --format markdown`. To specify a different file, use the `--out` option and specify the path and filename:

```
puppet strings generate --format markdown --out docs/INFO.md
```

View the Markdown file by opening it in a text editor or Markdown viewer.

Generate documentation in JSON

To generate reference documentation as JSON output to a file or to standard output, specify the `json` format when you run Strings.

Generate JSON output if you want to use the documentation in a custom application that reads JSON. By default, Strings prints JSON output to `STDOUT`. For details about Strings JSON output, see the [Strings JSON schema](#).

1. Change directory into the module: `cd /modules/<MODULE_NAME>`
2. Run the command: `puppet strings generate --format json`. To generate JSON documentation to a file instead, use the `--out` option and specify a filename:

```
puppet strings generate --format json --out documentation.json
```

Publish module documentation to GitHub Pages

To make your module documentation available on GitHub Pages, generate and publish HTML documentation with a Strings Rake task.

The `strings:gh_pages:update` Rake task is available in the `puppet-strings/tasks` directory. This Rake task keeps the `gh-pages` branch up to date with your current code, performing the following actions:

1. Creates a `doc` directory in the root of your project, if it doesn't already exist.
2. Creates a `gh-pages` branch of the current repository, if it doesn't already exist.
3. Checks out the `gh-pages` branch of the current repository.
4. Generates Strings HTML documentation.

5. Commits the documentation file and pushes it to the `gh-pages` branch with the `--force` flag.

To learn more about publishing on GitHub Pages, see the [GitHub Pages documentation](#).

1. If this is the first time you are running this task, you must first update your Gemfile and Rakefile.
 - a) Add the following to your Gemfile to use `puppet-strings`: `ruby gem 'puppet-strings'`
 - b) Add the following to your Rakefile to use the `puppet-strings` tasks: `ruby require 'puppet-strings/tasks'`

2. To generate, push, and publish your module's Strings documentation, run `strings:gh_pages:update`

The documentation is published after the task pushes the updated documentation to GitHub Pages.

Puppet Strings command reference

Modify the behavior of Puppet Strings by specifying command actions and options.

puppet strings command

Generates module documentation based on code and code comments. By default, running `puppet strings` generates HTML documentation for a module into a `./doc/` directory within that module.

To pass options or arguments, such as specifying Markdown or JSON output, use the `generate` action.

Usage:

```
puppet strings [--generate] [--server]
```

Action	Description
<code>generate</code>	Generates documentation with any specified parameters, including format and output location.
<code>server</code>	Serves documentation locally at <code>http://localhost:8808</code> 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
<code>--format</code>	Specifies a format for documentation.	Markdown, JSON	If not specified, outputs HTML documentation.

Option	Description	Values	Default
<code>--out</code>	Specifies an output location for documentation.	A valid directory location and filename.	If not specified, outputs to default locations depending on format: <ul style="list-style-type: none"> HTML: <code>./doc/</code> Markdown: main module directory) JSON: <code>STDOUT</code>
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.
<code>--debug, -d</code>	Logs debug information.	None.	If not specified, does not log debug information.
<code>--help</code>	Displays help documentation for the command.	None.	If specified, returns help information.
<code>--markup <FORMAT></code>	The markup format to use for documentation	<ul style="list-style-type: none"> "markdown" "textile" "rdoc" "ruby" "text" "html" "none" 	If no <code>--format</code> is specified, outputs HTML.
<code>--verbose, -v</code>	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
<code>--markup <FORMAT></code>	The markup format to use for documentation	<ul style="list-style-type: none"> "markdown" "textile" "rdoc" "ruby" "text" "html" "none" 	If no <code>--format</code> is specified, outputs HTML.
<code>--debug, -d</code>	Logs debug information.	None.	If not specified, does not log debug information.
<code>--help</code>	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.
<code>--modulepath</code>	Puppet option for setting the modulepath.	A valid path.	Defaults to the module path specified in the <code>puppet.conf</code> file.
<code>--verbose, -v</code>	Logs verbosely.	None.	If not specified, logs basic information.

Available Strings tags

@author

List the author or authors of a class, module, or method.

```
# @author Foo Bar
class MyClass; end
```

@api

Describes the resource as belonging to the private or public API. To mark a module element, such as a class, as private, specify as private:

```
# @api private
```

@example

Shows an example snippet of code for an object. The first line is an optional title, and any subsequent lines are automatically formatted as a code snippet. Use for specific examples of a given component. Use one example tag per example.

@param

Documents a parameter with a given name, type and optional description.

@!puppet.type.param

Documents dynamic type parameters. See the documenting resource types in the Strings [style guide](#) for detailed information.

@!puppet.type.property

Documents dynamic type properties. See the documenting resource types in the Strings [style guide](#) for detailed information.

@option

Used with a `@param` tag to defines what optional parameters the user can pass in an options hash to the method. For example:

```
# @param [Hash] opts
#   List of options
# @option opts [String] :option1
#   option 1 in the hash
# @option opts [Array] :option2
#   option 2 in the hash
```

@raise

Documents any exceptions that can be raised by the given component. For example:

```
# @raise PuppetError this error is raised if x
```

@return

Describes the return value (and type or types) of a method. You can list multiple return tags for a method if the method has distinct return cases. In this case, begin each case with "if". For example:

```
# An example 4.x function.
Puppet::Functions.create_function(:example) do
  # @param first The first parameter.
  # @param second The second parameter.
  # @return [String] If second argument is less than 10, the name of
  one item.
  # @return [Array] If second argument is greater than 10, a list of
  item names.
  # @example Calling the function.
  #   example('hi', 10)
  dispatch :example do
    param 'String', :first
    param 'Integer', :second
  end
  # ...
end
```

@see

Adds "see also" references. Accepts URLs or other code objects with an optional description at the end. The URL or object is automatically linked by YARD and does not need markup formatting. Appears in the generated documentation as a "See Also" section. Use one tag per reference, such as a website or related method.

@since

Lists the version in which the object was first added. Strings does not verify that the specified version exists. You are responsible for providing accurate information.

@summary

A description of the documented item, of 140 characters or fewer.

Puppet Strings style guide

To document your module with Puppet Strings, add descriptive tags and comments to your module code. Write consistent, clear code comments, and include at least basic information about each element of your module (such as classes or defined types).

Strings uses YARD-style tags and comments, along with the structure of the module code, to generate complete reference information for your module. Whenever you update your code, update your documentation comments at the same time.

This style guide applies to:

- Puppet Strings version 2.0 or later
- Puppet 4.0 or later

For information about the specific meaning of the terms 'must,' 'must not,' 'required,' 'should,' 'should not,' 'recommend,' 'may,' and 'optional,' see [RFC 2119](#).

The module README

In your module README, include basic module information and extended usage examples for common use cases. The README tells users what your module does and how to use it. Strings generates reference documentation, so typically, there is no need to include a reference section in your README. Strings generates information for type aliases or facts.

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 neselect_servers
#   Specifies one or more peers to not sync with. Puppet appends
#   'neselect' to each matching item in the `servers` array.
```

Example class

```
# @summary configures the Apache PHP module
#
# @example Basic usage
#   class { 'apache::mod::php':
#     package_name => 'mod_php5',
#     source       => '/etc/php/custom_config.conf',
#     php_version  => '7',
#   }
#
# @see http://php.net/manual/en/security.apache.php
#
# @param package_name
#   Names the package that installs mod_php
# @param package_ensure
#   Defines ensure for the PHP module package
# @param path
#   Defines the path to the mod_php shared object (.so) file.
# @param extensions
#   Defines an array of extensions to associate with PHP.
# @param content
#   Adds arbitrary content to php.conf.
# @param template
#   Defines the path to the php.conf template Puppet uses to generate the
#   configuration file.
# @param source
#   Defines the path to the default configuration. Values include a
#   puppet:/// path.
```

```
# @param root_group
#   Names a group with root access
# @param php_version
#   Names the PHP version Apache is using.
#
class apache::mod::php (
  $package_name      = undef,
  $package_ensure     = 'present',
  $path              = undef,
  Array $extensions   = ['.php'],
  $content            = undef,
  $template          = 'apache/mod/php.conf.erb',
  $source            = undef,
  $root_group        = $::apache::params::root_group,
  $php_version       = $::apache::params::php_version,
)
{
  ...
}
```

Example defined type

```
# @summary
#   Create and configure a MySQL database.
#
# @example Create a database
#   mysql::db { 'mydb':
#     user      => 'myuser',
#     password => 'mypass',
#     host      => 'localhost',
#     grant     => ['SELECT', 'UPDATE'],
#   }
#
# @param name
#   The name of the database to create. (dbname)
# @param user
#   The user for the database you're creating.
# @param password
#   The password for $user for the database you're creating.
# @param dbname
#   The name of the database to create.
# @param charset
#   The character set for the database.
# @param collate
#   The collation for the database.
# @param host
#   The host to use as part of user@host for grants.
# @param grant
#   The privileges to be granted for user@host on the database.
# @param sql
#   The path to the sqlfile you want to execute. This can be single file
#   specified as string, or it can be an array of strings.
# @param enforce_sql
#   Specifies whether to execute the sqlfiles on every run. If set to false,
#   sqlfiles runs only one time.
# @param ensure
#   Specifies whether to create the database. Valid values are 'present',
#   'absent'. Defaults to 'present'.
# @param import_timeout
#   Timeout, in seconds, for loading the sqlfiles. Defaults to 300.
# @param import_cat_cmd
```

```
# Command to read the sqlfile for importing the database. Useful for
compressed sqlfiles. For example, you can use 'zcat' for .gz files.
```

Functions

For custom Ruby functions, place documentation strings immediately before each dispatch call. For functions written in Puppet, place documentation strings immediately before the function name.

Include the following information for each function:

1. An untagged docstring describing what the function does.
2. One `@param` tag for each parameter in the function. See the parameters section for formatting guidelines.
3. A `@return` tag with the data type and a description of the returned value.
4. Optionally, a usage example, consisting of:
 - a. An `@example` tag with a description of a usage example on the same line.
 - b. A code example showing how the function is used. Place this example directly under the `@example` tag and description, indented two spaces.

Example Ruby function with one potential return type

```
# An example 4.x function.
Puppet::Functions.create_function(:example) do
  # @param first The first parameter.
  # @param second The second parameter.
  # @return [String] Returns a string.
  # @example Calling the function
  #   example('hi', 10)
  dispatch :example do
    param 'String', :first
    param 'Integer', :second
  end

  # ...
end
```

Example Ruby function with multiple potential return types

If the function has more than one potential return type, specify a `@return` tag for each. Begin each tag string with "if" to differentiate between cases.

```
# An example 4.x function.
Puppet::Functions.create_function(:example) do
  # @param first The first parameter.
  # @param second The second parameter.
  # @return [String] If second argument is less than 10, the name of one
  item.
  # @return [Array] If second argument is greater than 10, a list of item
  names.
  # @example Calling the function.
  #   example('hi', 10)
  dispatch :example do
    param 'String', :first
    param 'Integer', :second
  end

  # ...
end
```


Puppet function example

```
# @param name the name to say hello to.
# @return [String] Returns a string.
# @example Calling the function.
#   example('world')
function example(String $name) {
  "hello, $name"
}
```

Resource types

Add descriptions to the type and its attributes by passing either a here document (or "heredoc") or a short string to the `desc` method.

Strings automatically detects much of the information for types, including the parameters and properties, collectively known as attributes. To document the resource type itself, pass a heredoc to the `desc` method immediately after the type definition. Using a heredoc allows you to use multiple lines and Strings comment tags for your type documentation. For details about heredocs in Puppet, see the topic about [heredocs](#) in the language reference.

For attributes, where a short description is usually enough, pass a string to `desc` in the attribute. As with the `@param` tag, keep descriptions to 140 or fewer characters. If you need a longer description for an attribute, pass a heredoc to `desc` in the attribute itself.

You do not need to add tags for other method calls. Every other method call present in a resource type is automatically included and documented by Strings, and each attribute is updated accordingly in the final documentation. This includes method calls such as `defaultto`, `newvalue`, and `namevar`. If your type dynamically generates attributes, document those attributes with the `@!puppet.type.param` and `@!puppet.type.property` tags before the type definition. You may not use any other tags before the resource type definition.

Document the resource type description in the following order:

1. Directly under the type definition, indented two spaces, the `desc` method, with a heredoc including a descriptive delimiting keyword, such as `DESC`.
2. A `@summary` tag with a summary describing the type.
3. Optionally, usage examples, each consisting of:
 - a. An `@example` tag with a description of a usage example on the same line.
 - b. Code example showing how the type is used. Place this example directly under the `@example` tag and description, indented two spaces.

For types created with the resource API, follow the guidelines for standard resource types, but pass the heredoc or documentation string to a `desc` key in the data structure. You can include tags and multiple lines with the heredoc. Strings extracts the heredoc information along with other information from this data structure.

Example resource API type

The heredoc and documentation strings that Strings uses are called out in bold in this code example:

```
Puppet::ResourceApi.register_type(
  name: 'apt_key',
  docs: <<-EOS,
@summary Fancy new type.
@example Fancy new example.
  apt_key { '6F6B15509CF8E59E6E469F327F438280EF8D349F':
    source => 'http://apt.puppet.com/pubkey.gpg'
  }

```

This type provides Puppet with the capabilities to

manage GPG keys needed by apt to perform package validation. Apt has its own GPG keyring that can be manipulated through the ``apt-key`` command.

****Autorequires**:**

If Puppet is given the location of a key file which looks like an absolute path this type will autorequire that file.

EOS

```

  attributes: {
    ensure: {
      type: 'Enum[present, absent]',
      desc: 'Whether this apt key should be present or absent on the target
system.'**
    },
    id: {
      type: 'Variant[Pattern[/\A(0x)?[0-9a-fA-F]{8}\Z/], Pattern[/\A(0x)?[0-9a-fA-F]{16}\Z/], Pattern[/\A(0x)?[0-9a-fA-F]{40}\Z/]]',
      behaviour: :namevar,
      desc: 'The ID of the key you want to manage.',**
    },
    # ...
    created: {
      type: 'String',
      behavior: :read_only,
      desc: 'Date the key was created, in ISO format.',**
    },
  },
  autorequires: {
    file: '$source', # will evaluate to the value of the `source`
    attribute
    package: 'apt',
  },
)

```

Puppet tasks and plans

Strings documents Puppet tasks automatically, taking all information from the task metadata. Document task plans just as you would a class or defined type, with tags and descriptions in the plan file.

List the plan information in the following order:

1. A `@summary` tag, a space, and then a summary describing the plan.
2. Other tags such as `@see`, `@note`, or `@api private`.
3. Usage examples, each consisting of:
 - a. An `@example` tag with a description of a usage example on the same line.
 - b. Code example showing how the plan is used. Place this example directly under the `@example` tag and description, indented two spaces.
4. One `@param` tag for each parameter in the plan. See the parameters section for formatting guidelines. For example:

```

# @summary A simple plan.
#
# @param param1
#   First parameter description.
# @param param2
#   Second parameter description.
# @param param3
#   Third parameter description.
plan mymodule::my_plan(String $param1, $param2, Integer $param3 = 1) {
  run_task('mymodule::lb_remove', $param1, target => $param2)
}

```

Publishing modules

To share your module with other Puppet users, get contributions to your modules, and maintain your module releases, publish your module on the Puppet Forge. The Forge is a community repository of modules, written and contributed by open source Puppet and Puppet Enterprise users.

To publish your module, you'll:

1. Create a Forge account, if you don't already have one.
2. Prepare your module for packaging.
3. Add module metadata in the `metadata.json` file.
4. Build an uploadable tarball of your module.
5. Upload your module using the Forge web interface.

Naming your module

Your module has two names: a short name, like "mysql", and a long name that includes your Forge username, like "puppetlabs-mysql". When you upload your module to the Forge, use the module's long name.

Your module's short name is the same as that module's directory on your disk. This name must consist of letters, numbers, and underscores only; it can't contain dashes or periods.

The long name is composed of your Forge username and the short name of your module. For example, the "puppetlabs" user maintains a "mysql" module, which is located in a `./modules/mysql` directory and is known to the Forge as "puppetlabs-mysql".

In your module's `metadata.json` file, always use the long name of your module. This helps disambiguate modules that might have common short names, such as "mysql" or "apache." If you created your module with Puppet Development Kit (PDK), and you provided your Forge username to PDK, the `metadata.json` file already contains the correct long name for the module. Otherwise, edit your module's metadata with the correct long name.

Tip: Although the Forge expects to receive modules named `username-module`, its web interface presents them as `username/module`. Always use the `username-module` style in your metadata files and when issuing commands.

Related information

[Module metadata](#) on page 612

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 becomes part of your module long name, such as "bobcat-apache".
3. Check your email for a verification email from the Forge, and then follow the instructions in the email to verify your email address.

After you have verified your email address, you can publish modules to the Forge.

Prepare your module for publishing

Before you build your module package for publishing, make sure it's ready to be packaged.

Exclude unnecessary files from your package, remove or ignore any symlinks your module contains, and make sure your `metadata.json` file contains the correct information.

Tip: To publish your module to the Forge, your README, license file, changelog, and `metadata.json` must be UTF-8 encoded. If you used Puppet Development Kit (or the deprecated `puppet module generate` command) to create your module, these files are already UTF-8 encoded.

Excluding files from the package

To exclude certain files from your module build, include them in either an ignore file. Ignore files are useful for excluding files that are not needed to run the module, such as temporary files or files generated by spec tests. The ignore file must be in the root directory. You can use `.pdkignore`, `.gitignore`, or `.pmtignore` files in your module.

If you are building your module with PDK, your module package contains a `.pdkignore` file that already includes a list of commonly ignored files. To add or remove files to this list, define them in the module's `.sync.yml` file. For more information about customizing your module's configuration with `.sync.yml`, see the [PDK documentation](#).

If you are building your module with the `puppet module build` command, create a `.pmtignore` file and in it, list the files you want to exclude from the module package.

To prevent files, such as those in temporary directories, from ever being checked into your module's Git repo, list the files in a `.gitignore` file.

For example, a typical ignore file might look like this:

```
import/
/spec/fixtures/
.tmp
*.lock
*.local
.rbenv-gemsets
.ruby-version
build/
docs/
tests/
log/
junit/
tmp/
```

Removing symlinks from your module

Symlinks in modules are unsupported. If your module contains symlinks, either remove them or ignore them before you build your module.

If you try to build a module package that contains symlinks, you receive the following error:

```
Warning: Symlinks in modules are unsupported. Please investigate symlink
manifests/my-module.pp->manifests/init.pp.
Error: Found symlinks. Symlinks in modules are not allowed, please remove
them.
Error: Try 'puppet help module build' for usage
```

Verifying metadata

To publish your module on the Forge, it must contain required metadata in a `metadata.json` file. If you created your module with PDK or the deprecated `puppet module generate` command, you'll already have a `metadata.json` file. Open the file in any text editor, and make any necessary edits. For details on writing or editing the `metadata.json` file, see the related topic about module metadata.

Build a module package

To upload your module to the Forge, first build an uploadable module package with Puppet Development Kit.

PDK builds a `.tar.gz` package with the naming convention `<USERNAME>-<MODULE_SHORT_NAME>-<VERSION>.tar.gz`, in the module's `pkg/` subdirectory. For complete details about this task, see the PDK topic about [building module packages](#).

1. Change into the module directory by running `cd <MODULE_DIRECTORY>`
2. Build the package by running `pdk build`

3. Answer the question prompts as needed. You can use default answers to optional questions by pressing **Enter** at the prompt.
4. At the confirmation prompt, confirm or cancel package creation.

Upload a module to the Forge

To publish a new module release to the Forge, upload the module tarball using the web interface.

The module package must be a compiled `tar.gz` package of 10MB or less.

1. In your web browser, navigate to the Forge and log in.
2. Click **Publish** in the upper right hand corner of the screen.
3. On the upload page, click **Choose File** and use the file browser to locate and select the release tarball. Then click **Upload Release**.

After a successful upload, your browser loads the new release page for your module. If there were any errors on your upload, they appear on the same screen. Your module's README, Changelog, and License files are displayed on your module's Forge page.

Publish modules to the Forge with Travis CI

You can automatically publish new versions of your module to the Forge using Travis CI.

1. If this is your first time using Travis CI for automatic publishing, you must first enable Travis CI to publish to the Forge.
 - a) Enable Travis CI for the module repository.
 - b) Generate a Travis-encrypted Forge password string. For instructions, see the Travis CI [encryption keys docs](#).
 - c) Create a `.travis.yml` file in the module's repository base. Include a deployment section that includes your Forge username and the encrypted Forge password, such as:

```
deploy:
  provider: puppetforge
  user: <FORGE_USER>
  password:
    secure: "<ENCRYPTED_FORGE_PASSWORD>"
  on:
    tags: true
    # all_branches is required to use tags
    all_branches: true
```

2. To publish to the Forge with Travis CI, update, tag, and push your repository.
 - a) Update the version number in the module's `metadata.json` file and commit the change to the module repository.
 - b) Tag the module repo with the desired version number. For more information about how to do this, see Git docs on [basic tagging](#).
 - c) Push the commit and tag to your Git repository. Travis CI builds and publish the module.

Deprecate a module on the Forge

To let your module users know that you are no longer maintaining your module, deprecate your module on the Forge.

File a ticket in the [FORGE](#) project on the Puppet JIRA site. The ticket must include:

- The full name of the module to be deprecated, such as `puppetlabs-apache`.
- The reason for the deprecation. The reason is publicly displayed on the Forge.
- A recommended alternative module or workaround.

Delete a module release from the Forge

To delete a release of your module, use the Forge web interface. A deleted release is still downloadable via the Forge page or `puppet module` command if a user requests the module by specific version.

Restriction: You cannot delete a released version and then upload a new version of the same release.

1. In your web browser, navigate to the Forge and log in.
2. Click **Your Modules**.
3. Go to the page of the module release you want to delete.
4. Click **Select another release**, choose the release you want from the drop-down list, and click **Delete**.
5. On the confirmation page that loads, supply a reason for the deletion and submit.

The reason you give for deleting your module is visible to Forge users.

6. Click **Yes, delete it**.

On your module page, the confirmation of the deletion is displayed.

Related information

[Finding and downloading deleted modules](#) on page 600

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 the Contributor License Agreement when it comments on your pull request.

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 primary server branch, unless one of our module developers specifically asks you to base it on a different branch.

Remember: Never work directly on the primary server 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 [platform lifecycle](#) 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.

Designing system configs: roles and profiles

Your typical goal with Puppet is to build complete system configurations, which manage all of the software, services, and configuration that you care about on a given system. The roles and profiles method can help keep complexity under control and make your code more reusable, reconfigurable, and refactorable.

- [The roles and profiles method](#) on page 641

The *roles and profiles* method is the most reliable way to build reusable, configurable, and refactorable system configurations.

- [Roles and profiles example](#) on page 645

This example demonstrates a complete roles and profiles workflow. Use it to understand the roles and profiles method as a whole. Additional examples show how to design advanced configurations by refactoring this example code to a higher level of complexity.

- [Designing advanced profiles](#) on page 648

In this advanced example, we iteratively refactor our basic roles and profiles example to handle real-world concerns. The final result is — with only minor differences — the Jenkins profile we use in production here at Puppet.

- [Designing convenient roles](#) on page 664

There are several approaches to building roles, and you must decide which ones are most convenient for you and your team.

The roles and profiles method

The *roles and profiles* method is the most reliable way to build reusable, configurable, and refactorable system configurations.

It's not a straightforward recipe: you must think hard about the nature of your infrastructure and your team. It's also not a final state: expect to refine your configurations over time. Instead, it's an approach to *designing your infrastructure's interface* — sealing away incidental complexity, surfacing the significant complexity, and making sure your data behaves predictably.

Building configurations without roles and profiles

Without roles and profiles, people typically build system configurations in their node classifier or main manifest, using Hiera to handle tricky inheritance problems. A standard approach is to create a group of similar nodes and assign classes to it, then create child groups with extra classes for nodes that have additional needs. Another common pattern is to put everything in Hiera, using a very large hierarchy that reflects every variation in the infrastructure.

If this works for you, then it works! You might not need roles and profiles. But most people find direct building gets difficult to understand and maintain over time.

Configuring roles and profiles

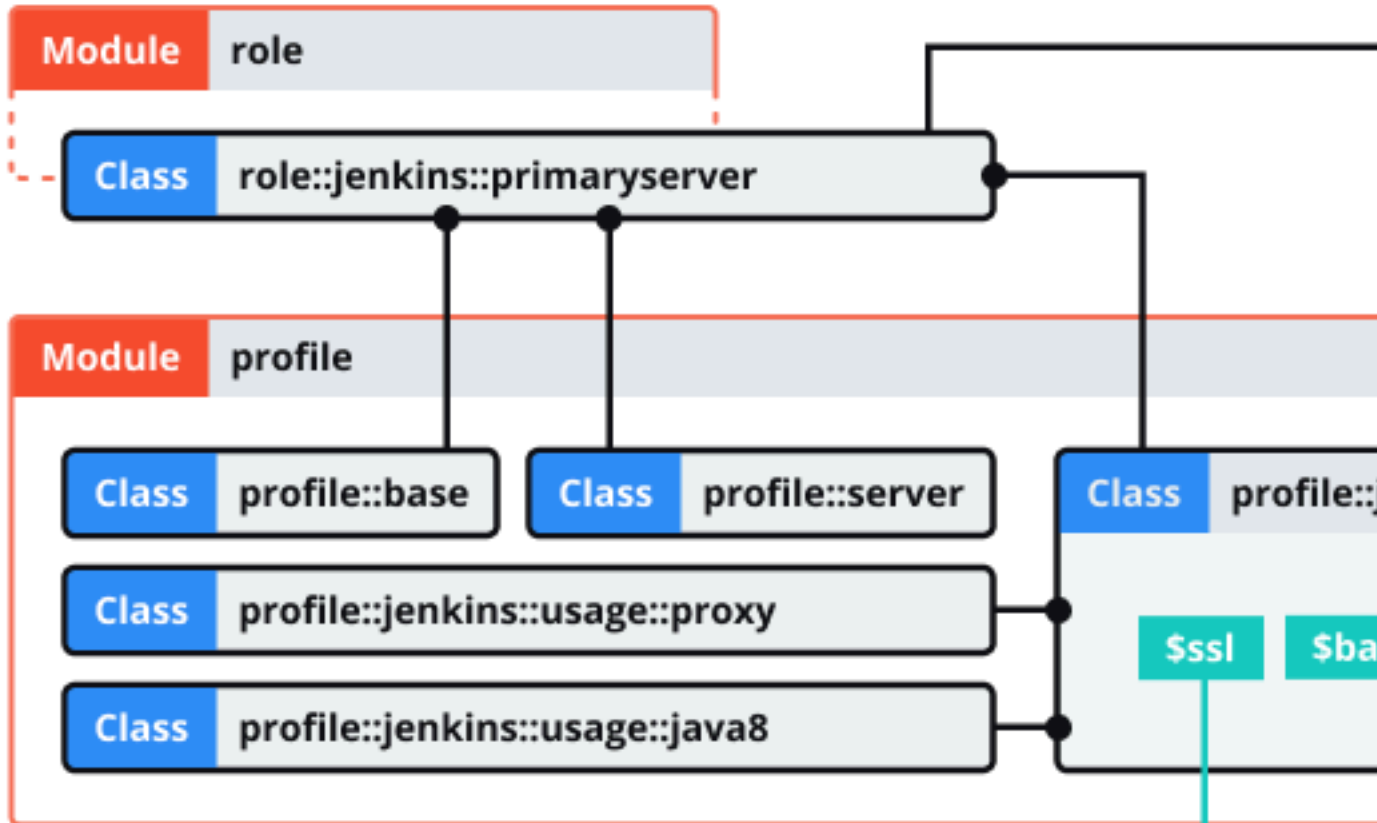
Roles and profiles are *two extra layers of indirection* between your node classifier and your component modules.

The roles and profiles method separates your code into three levels:

- Component modules — Normal modules that manage one particular technology, for example puppetlabs/apache.
- Profiles — Wrapper classes that use multiple component modules to configure a layered technology stack.
- Roles — Wrapper classes that use multiple profiles to build a complete system configuration.

These extra layers of indirection might seem like they add complexity, but they give you a space to build practical, business-specific interfaces to the configuration you care most about. A better interface makes hierarchical data easier to use, makes system configurations easier to read, and makes refactoring easier.

Node Classifier Group



```

Data groups/ci
(...) ::ssl= true

```

```

Data stages/dev
(...) ::backup= false

```

```

Data nodes/ci-primaryserver02-delivery.example.com
(...) ::site_alias= jenkins.example.com

```

In short, from top to bottom:

- Your node classifier assigns one *role* class to a group of nodes. The role manages a whole system configuration, so no other classes are needed. The node classifier does not configure the role in any way.
- That role class declares some *profile* classes with `include`, and does nothing else. For example:

```
class role::jenkins::master {
  include profile::base
  include profile::server
  include profile::jenkins::master
}
```

- Each profile configures a layered technology stack, using multiple component modules and the built-in resource types. (In the diagram, `profile::jenkins::master` uses `puppet/jenkins`, `puppetlabs/apt`, a home-built backup module, and some package and file resources.)
- Profiles can take configuration data from the console, Hiera, or Puppet lookup. (In the diagram, three different hierarchy levels contribute data.)
- Classes from component modules are always declared via a profile, and never assigned directly to a node.
 - If a component class has parameters, you specify them in the profile; never use Hiera or Puppet lookup to override component class params.

Rules for profile classes

There are rules for writing profile classes.

- Make sure you can safely `include` any profile multiple times — don't use resource-like declarations on them.
- Profiles can `include` other profiles.
- Profiles own all the class parameters for their component classes. If the profile omits one, that means you definitely want the default value; the component class shouldn't use a value from Hiera data. If you need to set a class parameter that was omitted previously, refactor the profile.
- There are three ways a profile can get the information it needs to configure component classes:
 - If your business always uses the same value for a given parameter, hardcode it.
 - If you can't hardcode it, try to compute it based on information you already have.
 - Finally, if you can't compute it, look it up in your data. To reduce lookups, identify cases where multiple parameters can be derived from the answer to a single question.

This is a game of trade-offs. Hardcoded parameters are the easiest to read, and also the least flexible. Putting values in your Hiera data is very flexible, but can be very difficult to read: you might have to look through a lot of files (or run a lot of lookup commands) to see what the profile is actually doing. Using conditional logic to derive a value is a middle-ground. Aim for the most readable option you can get away with.

Rules for role classes

There are rules for writing role classes.

- The only thing roles should do is declare profile classes with `include`. Don't declare any component classes or normal resources in a role.

Optionally, roles can use conditional logic to decide which profiles to use.

- Roles should not have any class parameters of their own.
- Roles should not set class parameters for any profiles. (Those are all handled by data lookup.)
- The name of a role should be based on your business's *conversational name* for the type of node it manages.

This means that if you regularly call a machine a "Jenkins master," it makes sense to write a role named `role::jenkins::master`. But if you call it a "web server," you shouldn't use a name like `role::nginx` — go with something like `role::web` instead.

Methods for data lookup

Profiles usually require some amount of configuration, and they must use data lookup to get it.

This profile uses the automatic class parameter lookup to request data.

```
# Example Hiera data
profile::jenkins::jenkins_port: 8000
profile::jenkins::java_dist: jre
profile::jenkins::java_version: '8'

# Example manifest
class profile::jenkins (
  Integer $jenkins_port,
  String  $java_dist,
  String  $java_version
) {
  # ...
}
```

This profile omits the parameters and uses the lookup function:

```
class profile::jenkins {
  $jenkins_port = lookup('profile::jenkins::jenkins_port', {value_type =>
    String, default_value => '9091'})
  $java_dist    = lookup('profile::jenkins::java_dist',      {value_type =>
    String, default_value => 'jdk'})
  $java_version = lookup('profile::jenkins::java_version', {value_type =>
    String, default_value => 'latest'})
  # ...
}
```

In general, class parameters are preferable to lookups. They integrate better with tools like Puppet strings, and they're a reliable and well-known place to look for configuration. But using `lookup` is a fine approach if you aren't comfortable with automatic parameter lookup. Some people prefer the full lookup key to be written in the profile, so they can globally grep for it.

Roles and profiles example

This example demonstrates a complete roles and profiles workflow. Use it to understand the roles and profiles method as a whole. Additional examples show how to design advanced configurations by refactoring this example code to a higher level of complexity.

Configure Jenkins master servers with roles and profiles

Jenkins is a continuous integration (CI) application that runs on the JVM. The Jenkins master server provides a web front-end, and also runs CI tasks at scheduled times or in reaction to events.

In this example, we manage the configuration of Jenkins master servers.

Set up your prerequisites

If you're new to using roles and profiles, do some additional setup before writing any new code.

1. Create two modules: one named `role`, and one named `profile`.

If you deploy your code with Code Manager or r10k, put these two modules in your control repository instead of declaring them in your Puppetfile, because Code Manager and r10k reserve the `modules` directory for their own use.

- a. Make a new directory in the repo named `site`.
- b. Edit the `environment.conf` file to add `site` to the `modulepath`. (For example: `modulepath = site:modules:$basemodulepath`).
- c. Put the `role` and `profile` modules in the `site` directory.

2. Make sure Hiera or Puppet lookup is set up and working, with a hierarchy that works well for you.

Choose component modules

For our example, we want to manage Jenkins itself using the `puppet/jenkins` module.

Jenkins requires Java, and the `puppet/jenkins` module can manage it automatically. But we want finer control over Java, so we're going to disable that. So, we need a Java module, and `puppetlabs/java` is a good choice.

That's enough to start with. We can refactor and expand when we have those working.

To learn more about these modules, see [puppet/jenkins](#) and [puppetlabs/java](#).

Write a profile

From a Puppet perspective, a profile is just a normal class stored in the `profile` module.

Make a new class called `profile::jenkins::master`, located at `.../profile/manifests/jenkins/master.pp`, and fill it with Puppet code.

```
# /etc/puppetlabs/code/environments/production/site/profile/manifests/
jenkins/master.pp
class profile::jenkins::master (
  String $jenkins_port = '9091',
  String $java_dist     = 'jdk',
  String $java_version  = 'latest',
) {

  class { ['jenkins']:
    configure_firewall => true,
    install_java       => false,
    port               => $jenkins_port,
    config_hash        => {
      'HTTP_PORT'   => { 'value' => $jenkins_port },
      'JENKINS_PORT' => { 'value' => $jenkins_port },
    },
  },

  class { ['java']:
    distribution => $java_dist,
    version      => $java_version,
    before       => Class['jenkins'],
  }
}
```

This is pretty simple, but is already benefiting us: our interface for configuring Jenkins has gone from 30 or so parameters on the Jenkins class (and many more on the Java class) down to three. Notice that we've hardcoded the `configure_firewall` and `install_java` parameters, and have reused the value of `$jenkins_port` in three places.

Related information

[Rules for profile classes](#) on page 644

There are rules for writing profile classes.

[Methods for data lookup](#) on page 645

Profiles usually require some amount of configuration, and they must use data lookup to get it.

Set data for the profile

Let's assume the following:

- We use some custom facts:
 - `group`: The group this node belongs to. (This is usually either a department of our business, or a large-scale function shared by many nodes.)
 - `stage`: The deployment stage of this node (dev, test, or prod).

- We have a five-layer hierarchy:
 - `console_data` for data defined in the console.
 - `nodes/{trusted.certname}` for per-node overrides.
 - `groups/{facts.group}/{facts.stage}` for setting stage-specific data within a group.
 - `groups/{facts.group}` for setting group-specific data.
 - `common` for global fallback data.
- We have a few one-off Jenkins masters, but most of them belong to the `ci` group.
- Our quality engineering department wants masters in the `ci` group to use the Oracle JDK, but one-off machines can just use the platform's default Java.
- QE also wants their prod masters to listen on port 80.

Set appropriate values in the data, using either Hiera or configuration data in the console.

```
# /etc/puppetlabs/code/environments/production/data/nodes/ci-
master01.example.com.yaml
# --Nothing. We don't need any per-node values right now.

# /etc/puppetlabs/code/environments/production/data/groups/ci/prod.yaml
profile::jenkins::master::jenkins_port: '80'

# /etc/puppetlabs/code/environments/production/data/groups/ci.yaml
profile::jenkins::master::java_dist: 'oracle-jdk8'
profile::jenkins::master::java_version: '8u92'

# /etc/puppetlabs/code/environments/production/data/common.yaml
# --Nothing. Just use the default parameter values.
```

Write a role

To write roles, we consider the machines we'll be managing and decide what else they need in addition to that Jenkins profile.

Our Jenkins masters don't serve any other purpose. But we have some profiles (code not shown) that we expect every machine in our fleet to have:

- `profile::base` must be assigned to every machine, including workstations. It manages basic policies, and uses some conditional logic to include OS-specific profiles as needed.
- `profile::server` must be assigned to every machine that provides a service over the network. It makes sure ops can log into the machine, and configures things like timekeeping, firewalls, logging, and monitoring.

So a role to manage one of our Jenkins masters should include those classes as well.

```
class role::jenkins::master {
  include profile::base
  include profile::server
  include profile::jenkins::master
}
```

Related information

[Rules for role classes](#) on page 644

There are rules for writing role classes.

Assign the role to nodes

Finally, we assign `role::jenkins::master` to every node that acts as a Jenkins master.

Puppet has several ways to assign classes to nodes, so use whichever tool you feel best fits your team. Your main choices are:

- The console node classifier, which lets you group nodes based on their facts and assign classes to those groups.
- The main manifest which can use node statements or conditional logic to assign classes.

- [Hiera](#) or Puppet lookup — Use [the lookup function](#) to do a unique array merge on a special `classes` key, and pass the resulting array to the `include` function.

```
# /etc/puppetlabs/code/environments/production/manifests/site.pp
lookup('classes', {merge => unique}).include
```

Designing advanced profiles

In this advanced example, we iteratively refactor our basic roles and profiles example to handle real-world concerns. The final result is — with only minor differences — the Jenkins profile we use in production here at Puppet.

Along the way, we explain our choices and point out some of the common trade-offs you encounter as you design your own profiles.

Here's the basic Jenkins profile we're starting with:

```
# /etc/puppetlabs/code/environments/production/site/profile/manifests/
jenkins/master.pp
class profile::jenkins::master (
  String $jenkins_port = '9091',
  String $java_dist     = 'jdk',
  String $java_version  = 'latest',
) {

  class { ['jenkins']:
    configure_firewall => true,
    install_java       => false,
    port               => $jenkins_port,
    config_hash        => {
      'HTTP_PORT'    => { 'value' => $jenkins_port },
      'JENKINS_PORT' => { 'value' => $jenkins_port },
    },
  },

  class { ['java']:
    distribution => $java_dist,
    version      => $java_version,
    before       => Class['jenkins'],
  },
}
```

Related information

[Rules for profile classes](#) on page 644

There are rules for writing profile classes.

First refactor: Split out Java

We want to manage Jenkins masters *and* Jenkins agent nodes. We won't cover agent profiles in detail, but the first issue we encountered is that they also need Java.

We could copy and paste the Java class declaration; it's small, so keeping multiple copies up-to-date might not be too burdensome. But instead, we decided to break Java out into a separate profile. This way we can manage it one time, then include the Java profile in both the agent and master profiles.

Note: This is a common trade-off. Keeping a chunk of code in only one place (often called the DRY — "don't repeat yourself" — principle) makes it more maintainable and less vulnerable to rot. But it has a cost: your individual profile classes become less readable, and you must view more files to see what a profile actually does. To reduce that readability cost, try to break code out in units that make inherent sense. In this case, the Java profile's job is simple enough to guess by its name — your colleagues don't have to read its code to know that it manages Java 8. Comments can also help.

First, decide how configurable Java needs to be on Jenkins machines. After looking at our past usage, we realized that we use only two options: either we install Oracle's Java 8 distribution, or we default to OpenJDK 7, which the Jenkins module manages. This means we can:

- Make our new Java profile really simple: hardcode Java 8 and take no configuration.
- Replace the two Java parameters from `profile::jenkins::master` with one Boolean parameter (whether to let Jenkins handle Java).

Note: This is rule 4 in action. We reduce our profile's configuration surface by combining multiple questions into one.

Here's the new parameter list:

```
class profile::jenkins::master (
  String $jenkins_port = '9091',
  Boolean $install_jenkins_java = true,
) { # ...
```

And here's how we choose which Java to use:

```
class { 'jenkins':
  configure_firewall => true,
  install_java       => $install_jenkins_java,    # <--- here
  port              => $jenkins_port,
  config_hash       => {
    'HTTP_PORT'    => { 'value' => $jenkins_port },
    'JENKINS_PORT' => { 'value' => $jenkins_port },
  },
}

# When not using the jenkins module's java version, install java8.
unless $install_jenkins_java { include profile::jenkins::usage::java8 }
```

And our new Java profile:

```
::jenkins::usage::java8
# Sets up java8 for Jenkins on Debian
#
class profile::jenkins::usage::java8 {
  motd::register { 'Java usage profile (profile::jenkins::usage::java8)':'' }

  # OpenJDK 7 is already managed by the Jenkins module.
  # ::jenkins::install_java or ::jenkins::agent::install_java should be
  # false to use this profile
  # this can be set through the class parameter $install_jenkins_java
  case $::osfamily {
    'debian': {
      class { 'java':
        distribution => 'oracle-jdk8',
        version      => '8u92',
      }

      package { 'tzdata-java':
        ensure => latest,
      }
    }
  }
  default: {
    notify { ["profile::jenkins::usage::java8 cannot set up JDK on
${::osfamily}"] }
  }
}
```

Diff of first refactor

```
@@ -1,13 +1,12 @@
# /etc/puppetlabs/code/environments/production/site/profile/manifests/
jenkins/master.pp
class profile::jenkins::master (
-   String $jenkins_port = '9091',
-   String $java_dist    = 'jdk',
-   String $java_version = 'latest',
+   String $jenkins_port = '9091',
+   Boolean $install_jenkins_java = true,
) {

    class { 'jenkins':
        configure_firewall => true,
-       install_java       => false,
+       install_java       => $install_jenkins_java,
        port               => $jenkins_port,
        config_hash        => {
            'HTTP_PORT'    => { 'value' => $jenkins_port },
@@ -15,9 +14,6 @@ class profile::jenkins::master (
    },
}

-   class { 'java':
-       distribution => $java_dist,
-       version      => $java_version,
-       before       => Class['jenkins'],
-   }
+   # When not using the jenkins module's java version, install java8.
+   unless $install_jenkins_java { include profile::jenkins::usage::java8 }
}
```

Second refactor: Manage the heap

At Puppet, we manage the Java heap size for the Jenkins app. Production servers didn't have enough memory for heavy use.

The Jenkins module has a `jenkins::sysconfig` defined type for managing system properties, so let's use it:

```
# Manage the heap size on the master, in MB.
if($::memorysize_mb =~ Number and $::memorysize_mb > 8192)
{
    # anything over 8GB we should keep max 4GB for OS and others
    $heap = sprintf('%.0f', $::memorysize_mb - 4096)
} else {
    # This is calculated as 50% of the total memory.
    $heap = sprintf('%.0f', $::memorysize_mb * 0.5)
}
# Set java params, like heap min and max sizes. See
# https://wiki.jenkins-ci.org/display/JENKINS/Features+controlled+by
+system+properties
jenkins::sysconfig { 'JAVA_ARGS':
    value => "-Xms${heap}m -Xmx${heap}m -Djava.awt.headless=true
-XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -
Dhudson.model.DirectoryBrowserSupport.CSP=\\\\"default-src 'self'; img-src
'self'; style-src 'self';\\\"",
}
```

Note: Rule 4 again — we couldn't hardcode this, because we have some smaller Jenkins masters that can't spare the extra memory. But because our production masters are always on more powerful machines, we can calculate the heap based on the machine's memory size, which we can access as a fact. This lets us avoid extra configuration.

Diff of second refactor

```
@@ -16,4 +16,20 @@ class profile::jenkins::master (
    # When not using the jenkins module's java version, install java8.
    unless $install_jenkins_java { include profile::jenkins::usage::java8 }
+
+ # Manage the heap size on the master, in MB.
+ if($::memorysize_mb =~ Number and $::memorysize_mb > 8192)
+ {
+     # anything over 8GB we should keep max 4GB for OS and others
+     $heap = sprintf('%.0f', $::memorysize_mb - 4096)
+ } else {
+     # This is calculated as 50% of the total memory.
+     $heap = sprintf('%.0f', $::memorysize_mb * 0.5)
+ }
+ # Set java params, like heap min and max sizes. See
+ # https://wiki.jenkins-ci.org/display/JENKINS/Features+controlled+by
+system+properties
+ jenkins::sysconfig { 'JAVA_ARGS':
+     value => "-Xms${heap}m -Xmx${heap}m -Djava.awt.headless=true
+ -XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -
+ Hudson.model.DirectoryBrowserSupport.CSP=\\\\"default-src 'self'; img-src
+ 'self'; style-src 'self';\\\\"\"",
+ }
+
+ }
```

Third refactor: Pin the version

We dislike surprise upgrades, so we pin Jenkins to a specific version. We do this with a direct package URL instead of by adding Jenkins to our internal package repositories. Your organization might choose to do it differently.

First, we add a parameter to control upgrades. Now we can set a new value in `.../data/groups/ci/dev.yaml` while leaving `.../data/groups/ci.yaml` alone — our dev machines get the new Jenkins version first, and we can ensure everything works as expected before upgrading our prod machines.

```
class profile::jenkins::master (
    Variant[String[1], Boolean] $direct_download = 'http://pkg.jenkins-ci.org/
    debian-stable/binary/jenkins_1.642.2_all.deb',
    # ...
) { # ...
```

Then, we set the necessary parameters in the Jenkins class:

```
class { 'jenkins':
    lts                => true,                # <-- here
    repo               => true,                # <-- here
    direct_download    => $direct_download,    # <-- here
    version            => 'latest',            # <-- here
    service_enable     => true,
    service_ensure     => running,
    configure_firewall => true,
    install_java       => $install_jenkins_java,
    port               => $jenkins_port,
    config_hash        => {
        'HTTP_PORT'    => { 'value' => $jenkins_port },
        'JENKINS_PORT' => { 'value' => $jenkins_port },
    },
}
```

This was a good time to explicitly manage the Jenkins *service*, so we did that as well.

Diff of third refactor

```
@@ -1,10 +1,17 @@
# /etc/puppetlabs/code/environments/production/site/profile/manifests/
jenkins/master.pp
class profile::jenkins::master (
-   String $jenkins_port = '9091',
-   Boolean $install_jenkins_java = true,
+   String $jenkins_port = '9091',
+   Variant[String[1], Boolean] $direct_download = 'http://pkg.jenkins-
ci.org/debian-stable/binary/jenkins_1.642.2_all.deb',
+   Boolean $install_jenkins_java = true,
) {

    class { 'jenkins':
+       lts                => true,
+       repo               => true,
+       direct_download    => $direct_download,
+       version            => 'latest',
+       service_enable     => true,
+       service_ensure     => running,
+       configure_firewall => true,
+       install_java       => $install_jenkins_java,
+       port               => $jenkins_port,
```

Fourth refactor: Manually manage the user account

We manage a lot of user accounts in our infrastructure, so we handle them in a unified way. The `profile::server` class pulls in `virtual::users`, which has a lot of virtual resources we can selectively realize depending on who needs to log into a given machine.

Note: This has a cost — it's action at a distance, and you need to read more files to see which users are enabled for a given profile. But we decided the benefit was worth it: because all user accounts are written in one or two files, it's easy to see all the users that might exist, and ensure that they're managed consistently.

We're accepting difficulty in one place (where we can comfortably handle it) to banish difficulty in another place (where we worry it would get out of hand). Making this choice required that we know our colleagues and their comfort zones, and that we know the limitations of our existing code base and supporting services.

So, for this example, we change the Jenkins profile to work the same way; we manage the `jenkins` user alongside the rest of our user accounts. While we're doing that, we also manage a few directories that can be problematic depending on how Jenkins is packaged.

Some values we need are used by Jenkins agents as well as masters, so we're going to store them in a `params` class, which is a class that sets shared variables and manages no resources. This is a heavyweight solution, so wait until it provides real value before using it. In our case, we had a lot of OS-specific agent profiles (not shown in these examples), and they made a `params` class worthwhile.

Note: Just as before, "don't repeat yourself" is in tension with "keep it readable." Find the balance that works for you.

```
# We rely on virtual resources that are ultimately declared by
profile::server.
include profile::server

# Some default values that vary by OS:
include profile::jenkins::params
$jenkins_owner      = $profile::jenkins::params::jenkins_owner
$jenkins_group      = $profile::jenkins::params::jenkins_group
$master_config_dir  = $profile::jenkins::params::master_config_dir

file { ['/var/run/jenkins': ensure => 'directory' }
```

```

# Because our account::user class manages the '${master_config_dir}'
directory
# as the 'jenkins' user's homedir (as it should), we need to manage
# '${master_config_dir}/plugins` here to prevent the upstream
# rtyler-jenkins module from trying to manage the homedir as the config
# dir. For more info, see the upstream module's `manifests/plugin.pp`
# manifest.
file { ["${master_config_dir}/plugins":
  ensure => directory,
  owner   => $jenkins_owner,
  group   => $jenkins_group,
  mode    => '0755',
  require => [Group[$jenkins_group], User[$jenkins_owner]],
}]

Account::User <| tag == 'jenkins' |>

class { 'jenkins':
  lts           => true,
  repo          => true,
  direct_download => $direct_download,
  version       => 'latest',
  service_enable => true,
  service_ensure => running,
  configure_firewall => true,
  install_java   => $install_jenkins_java,
  manage_user    => false,                # <-- here
  manage_group   => false,                # <-- here
  manage_datadirs => false,                # <-- here
  port           => $jenkins_port,
  config_hash    => {
    'HTTP_PORT'   => { 'value' => $jenkins_port },
    'JENKINS_PORT' => { 'value' => $jenkins_port },
  },
}

```

Three things to notice in the code above:

- We manage users with a homegrown `account::user` defined type, which declares a user resource plus a few other things.
- We use an `Account::User` resource collector to realize the Jenkins user. This relies on `profile::server` being declared.
- We set the Jenkins class's `manage_user`, `manage_group`, and `manage_datadirs` parameters to false.
- We're now explicitly managing the `plugins` directory and the `run` directory.

Diff of fourth refactor

```

@@ -5,6 +5,33 @@ class profile::jenkins::master (
  Boolean                               $install_jenkins_java = true,
) {

+ # We rely on virtual resources that are ultimately declared by
+ profile::server.
+ include profile::server
+
+ # Some default values that vary by OS:
+ include profile::jenkins::params
+ $jenkins_owner      = $profile::jenkins::params::jenkins_owner
+ $jenkins_group      = $profile::jenkins::params::jenkins_group
+ $master_config_dir  = $profile::jenkins::params::master_config_dir
+
+ file { ['/var/run/jenkins': ensure => 'directory' }

```

```

+
+ # Because our account::user class manages the '${master_config_dir}'
+ directory
+ # as the 'jenkins' user's homedir (as it should), we need to manage
+ # `${master_config_dir}/plugins` here to prevent the upstream
+ # rtyler-jenkins module from trying to manage the homedir as the config
+ # dir. For more info, see the upstream module's `manifests/plugin.pp`
+ # manifest.
+ file { "${master_config_dir}/plugins":
+   ensure => directory,
+   owner  => $jenkins_owner,
+   group  => $jenkins_group,
+   mode   => '0755',
+   require => [Group[$jenkins_group], User[$jenkins_owner]],
+ }
+
+ Account::User <| tag == 'jenkins' |>
+
+ class { 'jenkins':
+   lts           => true,
+   repo          => true,
+@@ -14,6 +41,9 @@ class profile::jenkins::master (
+   service_ensure => running,
+   configure_firewall => true,
+   install_java   => $install_jenkins_java,
+ + manage_user     => false,
+ + manage_group    => false,
+ + manage_datadirs => false,
+   port            => $jenkins_port,
+   config_hash     => {
+     'HTTP_PORT'   => { 'value' => $jenkins_port },

```

Fifth refactor: Manage more dependencies

Jenkins always needs Git installed (because we use Git for source control at Puppet), and it needs SSH keys to access private Git repos and run commands on Jenkins agent nodes. We also have a standard list of Jenkins plugins we use, so we manage those too.

Managing Git is pretty easy:

```

package { 'git':
  ensure => present,
}

```

SSH keys are less easy, because they are sensitive content. We can't check them into version control with the rest of our Puppet code, so we put them in a custom mount point on one specific Puppet server.

Because this server is different from our normal Puppet servers, we made a rule about accessing it: you must look up the hostname from data instead of hardcoding it. This lets us change it in only one place if the secure server ever moves.

```

$secure_server = lookup('puppetlabs::ssl::secure_server')

file { "${master_config_dir}/.ssh":
  ensure => directory,
  owner  => $jenkins_owner,
  group  => $jenkins_group,
  mode   => '0700',
}

file { "${master_config_dir}/.ssh/id_rsa":
  ensure => file,
  owner  => $jenkins_owner,

```

```

    group => $jenkins_group,
    mode  => '0600',
    source => "puppet://${secure_server}/secure/delivery/id_rsa-jenkins",
  }

  file { ["${master_config_dir}/.ssh/id_rsa.pub":
    ensure => file,
    owner  => $jenkins_owner,
    group  => $jenkins_group,
    mode   => '0640',
    source => "puppet://${secure_server}/secure/delivery/id_rsa-
jenkins.pub",
  ]
}

```

Plugins are also a bit tricky, because we have a few Jenkins masters where we want to manually configure plugins. So we put the base list in a separate profile, and use a parameter to control whether we use it.

```

class profile::jenkins::master (
  Boolean                                $manage_plugins = false,
  # ...
) {
  # ...
  if $manage_plugins {
    include profile::jenkins::master::plugins
  }
}

```

In the plugins profile, we can use the `jenkins::plugin` resource type provided by the Jenkins module.

```

# /etc/puppetlabs/code/environments/production/site/profile/manifests/
jenkins/master/plugins.pp
class profile::jenkins::master::plugins {
  jenkins::plugin { 'audit2db': }
  jenkins::plugin { 'credentials': }
  jenkins::plugin { 'jquery': }
  jenkins::plugin { 'job-import-plugin': }
  jenkins::plugin { 'ldap': }
  jenkins::plugin { 'mailer': }
  jenkins::plugin { 'metadata': }
  # ... and so on.
}

```

Diff of fifth refactor

```

@@ -1,6 +1,7 @@
# /etc/puppetlabs/code/environments/production/site/profile/manifests/
jenkins/master.pp
class profile::jenkins::master (
  String                                $jenkins_port = '9091',
+ Boolean                                $manage_plugins = false,
  Variant[String[1], Boolean] $direct_download = 'http://pkg.jenkins-
ci.org/debian-stable/binary/jenkins_1.642.2_all.deb',
  Boolean                                $install_jenkins_java = true,
) {
@@ -14,6 +15,20 @@ class profile::jenkins::master (
  $jenkins_group      = $profile::jenkins::params::jenkins_group
  $master_config_dir  = $profile::jenkins::params::master_config_dir

+ if $manage_plugins {
+   # About 40 jenkins::plugin resources:
+   include profile::jenkins::master::plugins
+ }
}

```

```

+
+ # Sensitive info (like SSH keys) isn't checked into version control like
the
+ # rest of our modules; instead, it's served from a custom mount point on
a
+ # designated server.
+ $secure_server = lookup('puppetlabs::ssl::secure_server')
+
+ package { 'git':
+   ensure => present,
+ }
+
+ file { ['/var/run/jenkins': ensure => 'directory' }

+ # Because our account::user class manages the '${master_config_dir}'
directory
@@ -69,4 +84,29 @@ class profile::jenkins::master (
+   value => "-Xms${heap}m -Xmx${heap}m -Djava.awt.headless=true
-XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -
Dhudson.model.DirectoryBrowserSupport.CSP=\\\\"default-src 'self'; img-src
'self'; style-src 'self';\\\\"\"",
+ }

+ # Deploy the SSH keys that Jenkins needs to manage its agent machines and
+ # access Git repos.
+ file { ["${master_config_dir}/.ssh":
+   ensure => directory,
+   owner   => $jenkins_owner,
+   group   => $jenkins_group,
+   mode    => '0700',
+ }
+
+ file { ["${master_config_dir}/.ssh/id_rsa":
+   ensure => file,
+   owner   => $jenkins_owner,
+   group   => $jenkins_group,
+   mode    => '0600',
+   source  => "puppet://${secure_server}/secure/delivery/id_rsa-jenkins",
+ }
+
+ file { ["${master_config_dir}/.ssh/id_rsa.pub":
+   ensure => file,
+   owner   => $jenkins_owner,
+   group   => $jenkins_group,
+   mode    => '0640',
+   source  => "puppet://${secure_server}/secure/delivery/id_rsa-
jenkins.pub",
+ }
+
+ }

```

Sixth refactor: Manage logging and backups

Backing up is usually a good idea.

We can use our homegrown backup module, which provides a `backup::job` resource type (profile::server takes care of its prerequisites). But we should make backups optional, so people don't accidentally post junk to our backup server if they're setting up an ephemeral Jenkins instance to test something.

```

class profile::jenkins::master (
  Boolean                                $backups_enabled = false,
  # ...
) {
  # ...

```



```

    if $backups_enabled {
      backup::job { "jenkins-data-${::hostname}":
        files => $master_config_dir,
      }
    }
  }
}

```

Also, our teams gave us some conflicting requests for Jenkins logs:

- Some people want it to use syslog, like most other services.
- Others want a distinct log file so syslog doesn't get spammed, and they want the file to rotate more quickly than it does by default.

That implies a new parameter. We can make one called `$jenkins_logs_to_syslog` and default it to `undef`. If you set it to a standard syslog facility (like `daemon.info`), Jenkins logs there instead of its own file.

We use `jenkins::sysconfig` and our homegrown `logrotate::job` to do the work:

```

class profile::jenkins::master (
  Optional[String[1]] $jenkins_logs_to_syslog = undef,
  # ...
) {
  # ...
  if $jenkins_logs_to_syslog {
    jenkins::sysconfig { 'JENKINS_LOG':
      value => "$jenkins_logs_to_syslog",
    }
  }
  # ...
  logrotate::job { 'jenkins':
    log      => '/var/log/jenkins/jenkins.log',
    options => [
      'daily',
      'copytruncate',
      'missingok',
      'rotate 7',
      'compress',
      'delaycompress',
      'notifempty'
    ],
  }
}

```

Diff of sixth refactor

```

@@ -1,8 +1,10 @@
# /etc/puppetlabs/code/environments/production/site/profile/manifests/
jenkins/master.pp
class profile::jenkins::master (
  String                                $jenkins_port = '9091',
+ Boolean                              $backups_enabled = false,
  Boolean                              $manage_plugins = false,
  Variant[String[1], Boolean] $direct_download = 'http://pkg.jenkins-
ci.org/debian-stable/binary/jenkins_1.642.2_all.deb',
+ Optional[String[1]]            $jenkins_logs_to_syslog = undef,
  Boolean                              $install_jenkins_java = true,
) {

@@ -84,6 +86,15 @@ class profile::jenkins::master (
  value => "-Xms${heap}m -Xmx${heap}m -Djava.awt.headless=true
-XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -

```

```

Dhudson.model.DirectoryBrowserSupport.CSP=\\\\"default-src 'self'; img-src
'self'; style-src 'self';\\\\"";
}

+ # Forward jenkins master logs to syslog.
+ # When set to facility.level the jenkins_log uses that value instead of a
+ # separate log file, for example daemon.info
+ if $jenkins_logs_to_syslog {
+   jenkins::sysconfig { 'JENKINS_LOG':
+     value => "$jenkins_logs_to_syslog",
+   }
+ }
+
+ # Deploy the SSH keys that Jenkins needs to manage its agent machines and
+ # access Git repos.
+ file { ["${master_config_dir}/.ssh":
@@ -109,4 +120,29 @@ class profile::jenkins::master (
+   source => "puppet://${secure_server}/secure/delivery/id_rsa-
jenkins.pub",
+ }

+ # Back up Jenkins' data.
+ if $backups_enabled {
+   backup::job { "jenkins-data-${::hostname}":
+     files => $master_config_dir,
+   }
+ }

+ # (QENG-1829) Logrotate rules:
+ # Jenkins' default logrotate config retains too much data: by default, it
+ # rotates jenkins.log weekly and retains the last 52 weeks of logs.
+ # Considering we almost never look at the logs, let's rotate them daily
+ # and discard after 7 days to reduce disk usage.
+ logrotate::job { 'jenkins':
+   log      => '/var/log/jenkins/jenkins.log',
+   options => [
+     'daily',
+     'copytruncate',
+     'missingok',
+     'rotate 7',
+     'compress',
+     'delaycompress',
+     'notifempty'
+   ],
+ }
+
+ }

```

Seventh refactor: Use a reverse proxy for HTTPS

We want the Jenkins web interface to use HTTPS, which we can accomplish with an Nginx reverse proxy. We also want to standardize the ports: the Jenkins app always binds to its default port, and the proxy always serves over 443 for HTTPS and 80 for HTTP.

If we want to keep vanilla HTTP available, we can provide an `$ssl` parameter. If set to `false` (the default), you can access Jenkins via both HTTP and HTTPS. We can also add a `$site_alias` parameter, so the proxy can listen on a hostname other than the node's main FQDN.

```

class profile::jenkins::master (
  Boolean          $ssl = false,
  Optional[String[1]] $site_alias = undef,
  # IMPORTANT: notice that $jenkins_port is removed.
  # ...

```

Set `configure_firewall => false` in the Jenkins class:

```
class { 'jenkins':
  lts           => true,
  repo         => true,
  direct_download => $direct_download,
  version      => 'latest',
  service_enable => true,
  service_ensure => running,
  configure_firewall => false,          # <-- here
  install_java  => $install_jenkins_java,
  manage_user   => false,
  manage_group  => false,
  manage_datadirs => false,
  # IMPORTANT: notice that port and config_hash are removed.
}
```

We need to deploy SSL certificates where Nginx can reach them. Because we serve a lot of things over HTTPS, we already had a profile for that:

```
# Deploy the SSL certificate/chain/key for sites on this domain.
include profile::ssl::delivery_wildcard
```

This is also a good time to add some info for the message of the day, handled by puppetlabs/motd:

```
motd::register { 'Jenkins CI master (profile::jenkins::master)': }

if $site_alias {
  motd::register { 'jenkins-site-alias':
    content => @("END"),
              profile::jenkins::master::proxy

              Jenkins site alias: ${site_alias}
              |-END
    order   => 25,
  }
}
```

The bulk of the work is handled by a new profile called `profile::jenkins::master::proxy`. We're omitting the code for brevity; in summary, what it does is:

- Include `profile::nginx`.
- Use resource types from the `jfryman/nginx` to set up a vhost, and to force a redirect to HTTPS if we haven't enabled vanilla HTTP.
- Set up logstash forwarding for access and error logs.
- Include `profile::fw::https` to manage firewall rules, if necessary.

Then, we declare that profile in our main profile:

```
class { 'profile::jenkins::master::proxy':
  site_alias => $site_alias,
  require_ssl => $ssl,
}
```

Important:

We are now breaking rule 1, the most important rule of the roles and profiles method. Why?

Because `profile::jenkins::master::proxy` is a "private" profile that belongs solely to `profile::jenkins::master`. It will never be declared by any role or any other profile.

This is the only exception to rule 1: if you're separating out code *for the sole purpose of readability* --- that is, if you could paste the private profile's contents into the main profile for the exact same effect --- you can use a resource-like declaration on the private profile. This lets you consolidate your data lookups and make the private profile's inputs more visible, while keeping the main profile a little cleaner. If you do this, you must make sure to document that the private profile is private.

If there is any chance that this code might be reused by another profile, obey rule 1.

Diff of seventh refactor

```
@@ -1,8 +1,9 @@
# /etc/puppetlabs/code/environments/production/site/profile/manifests/
jenkins/master.pp
class profile::jenkins::master (
- String                                $jenkins_port = '9091',
  Boolean                               $backups_enabled = false,
  Boolean                               $manage_plugins = false,
+ Boolean                               $ssl = false,
+ Optional[String[1]]                  $site_alias = undef,
  Variant[String[1], Boolean] $direct_download = 'http://pkg.jenkins-
ci.org/debian-stable/binary/jenkins_1.642.2_all.deb',
  Optional[String[1]]                  $jenkins_logs_to_syslog = undef,
  Boolean                               $install_jenkins_java = true,
@@ -11,6 +12,9 @@ class profile::jenkins::master (
# We rely on virtual resources that are ultimately declared by
profile::server.
include profile::server

+ # Deploy the SSL certificate/chain/key for sites on this domain.
+ include profile::ssl::delivery_wildcard
+
+ # Some default values that vary by OS:
+ include profile::jenkins::params
+ $jenkins_owner = $profile::jenkins::params::jenkins_owner
@@ -22,6 +26,31 @@ class profile::jenkins::master (
include profile::jenkins::master::plugins
}

+ motd::register { 'Jenkins CI master (profile::jenkins::master)': }
+
+ # This adds the site_alias to the message of the day for convenience when
+ # logging into a server via FQDN. Because of the way motd::register
+ # works, we
+ # need a sort of funny formatting to put it at the end (order => 25) and
+ # to
+ # list a class so there isn't a random "--" at the end of the message.
+ if $site_alias {
+   motd::register { 'jenkins-site-alias':
+     content => @("END"),
+     profile::jenkins::master::proxy
+
+     Jenkins site alias: ${site_alias}
+     |-END
+     order    => 25,
+   }
+ }
+
+ # This is a "private" profile that sets up an Nginx proxy -- it's only
+ # ever
+ # declared in this class, and it would work identically pasted inline.
+ # But because it's long, this class reads more cleanly with it separated
+ # out.
```

```

+ class { 'profile::jenkins::master::proxy':
+   site_alias => $site_alias,
+   require_ssl => $ssl,
+ }
+
# Sensitive info (like SSH keys) isn't checked into version control like
the
# rest of our modules; instead, it's served from a custom mount point on
a
# designated server.
@@ -56,16 +85,11 @@ class profile::jenkins::master (
    version          => 'latest',
    service_enable   => true,
    service_ensure   => running,
-   configure_firewall => true,
+   configure_firewall => false,
    install_java     => $install_jenkins_java,
    manage_user      => false,
    manage_group     => false,
    manage_datadirs  => false,
-   port             => $jenkins_port,
-   config_hash      => {
-     'HTTP_PORT'    => { 'value' => $jenkins_port },
-     'JENKINS_PORT' => { 'value' => $jenkins_port },
-   },
- }

# When not using the jenkins module's java version, install java8.

```

The final profile code

After all of this refactoring (and a few more minor adjustments), here's the final code for `profile::jenkins::master`.

```

# /etc/puppetlabs/code/environments/production/site/profile/manifests/
jenkins/master.pp
# Class: profile::jenkins::master
#
# Install a Jenkins master that meets Puppet's internal needs.
#
class profile::jenkins::master (
  Boolean          $backups_enabled = false,
  Boolean          $manage_plugins = false,
  Boolean          $ssl = false,
  Optional[String[1]] $site_alias = undef,
  Variant[String[1], Boolean] $direct_download = 'http://pkg.jenkins-ci.org/
debian-stable/binary/jenkins_1.642.2_all.deb',
  Optional[String[1]] $jenkins_logs_to_syslog = undef,
  Boolean          $install_jenkins_java = true,
) {

  # We rely on virtual resources that are ultimately declared by
  profile::server.
  include profile::server

  # Deploy the SSL certificate/chain/key for sites on this domain.
  include profile::ssl::delivery_wildcard

  # Some default values that vary by OS:
  include profile::jenkins::params
  $jenkins_owner      = $profile::jenkins::params::jenkins_owner
  $jenkins_group      = $profile::jenkins::params::jenkins_group
  $master_config_dir  = $profile::jenkins::params::master_config_dir

```

```

if $manage_plugins {
  # About 40 jenkins::plugin resources:
  include profile::jenkins::master::plugins
}

motd::register { 'Jenkins CI master (profile::jenkins::master)': }

# This adds the site_alias to the message of the day for convenience when
# logging into a server via FQDN. Because of the way motd::register works,
we
# need a sort of funny formatting to put it at the end (order => 25) and
to
# list a class so there isn't a random "---" at the end of the message.
if $site_alias {
  motd::register { 'jenkins-site-alias':
    content => @("END"),
    profile::jenkins::master::proxy

    Jenkins site alias: ${site_alias}
    |-END
    order    => 25,
  }
}

# This is a "private" profile that sets up an Nginx proxy -- it's only
ever
# declared in this class, and it would work identically pasted inline.
# But because it's long, this class reads more cleanly with it separated
out.
class { 'profile::jenkins::master::proxy':
  site_alias => $site_alias,
  require_ssl => $ssl,
}

# Sensitive info (like SSH keys) isn't checked into version control like
the
# rest of our modules; instead, it's served from a custom mount point on a
# designated server.
$secure_server = lookup('puppetlabs::ssl::secure_server')

# Dependencies:
# - Pull in apt if we're on Debian.
# - Pull in the 'git' package, used by Jenkins for Git polling.
# - Manage the 'run' directory (fix for busted Jenkins packaging).
if $::osfamily == 'Debian' { include apt }

package { 'git':
  ensure => present,
}

file { ['/var/run/jenkins': ensure => 'directory' }

# Because our account::user class manages the '${master_config_dir}'
directory
# as the 'jenkins' user's homedir (as it should), we need to manage
# `${master_config_dir}/plugins` here to prevent the upstream
# rtyler-jenkins module from trying to manage the homedir as the config
# dir. For more info, see the upstream module's `manifests/plugin.pp`
# manifest.
file { "${master_config_dir}/plugins":
  ensure => directory,
  owner  => $jenkins_owner,
  group  => $jenkins_group,
}

```

```

    mode      => '0755',
    require => [Group[$jenkins_group], User[$jenkins_owner]],
  }

Account::User <| tag == 'jenkins' |>

class { 'jenkins':
  lts           => true,
  repo         => true,
  direct_download => $direct_download,
  version      => 'latest',
  service_enable => true,
  service_ensure => running,
  configure_firewall => false,
  install_java  => $install_jenkins_java,
  manage_user   => false,
  manage_group  => false,
  manage_datadirs => false,
}

# When not using the jenkins module's java version, install java8.
unless $install_jenkins_java { include profile::jenkins::usage::java8 }

# Manage the heap size on the master, in MB.
if($::memorysize_mb =~ Number and $::memorysize_mb > 8192)
{
  # anything over 8GB we should keep max 4GB for OS and others
  $heap = sprintf('%.0f', $::memorysize_mb - 4096)
} else {
  # This is calculated as 50% of the total memory.
  $heap = sprintf('%.0f', $::memorysize_mb * 0.5)
}
# Set java params, like heap min and max sizes. See
# https://wiki.jenkins-ci.org/display/JENKINS/Features+controlled+by
+system+properties
jenkins::sysconfig { 'JAVA_ARGS':
  value => "-Xms${heap}m -Xmx${heap}m -Djava.awt.headless=true
-XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -
Dhudson.model.DirectoryBrowserSupport.CSP=\\\\"default-src 'self'; img-src
'self'; style-src 'self';\\\"",
}

# Forward jenkins master logs to syslog.
# When set to facility.level the jenkins_log uses that value instead of a
# separate log file, for example daemon.info
if $jenkins_logs_to_syslog {
  jenkins::sysconfig { 'JENKINS_LOG':
    value => "$jenkins_logs_to_syslog",
  }
}

# Deploy the SSH keys that Jenkins needs to manage its agent machines and
# access Git repos.
file { "${master_config_dir}/.ssh":
  ensure => directory,
  owner  => $jenkins_owner,
  group  => $jenkins_group,
  mode   => '0700',
}

file { "${master_config_dir}/.ssh/id_rsa":
  ensure => file,
  owner  => $jenkins_owner,
  group  => $jenkins_group,

```

```

    mode    => '0600',
    source => "puppet://${secure_server}/secure/delivery/id_rsa-jenkins",
  }

  file { ["${master_config_dir}/.ssh/id_rsa.pub":
    ensure => file,
    owner  => $jenkins_owner,
    group  => $jenkins_group,
    mode   => '0640',
    source => "puppet://${secure_server}/secure/delivery/id_rsa-
jenkins.pub",
  ]

  # Back up Jenkins' data.
  if $backups_enabled {
    backup::job { ["jenkins-data-${::hostname}":
      files => $master_config_dir,
    ]
  }

  # (QENG-1829) Logrotate rules:
  # Jenkins' default logrotate config retains too much data: by default, it
  # rotates jenkins.log weekly and retains the last 52 weeks of logs.
  # Considering we almost never look at the logs, let's rotate them daily
  # and discard after 7 days to reduce disk usage.
  logrotate::job { 'jenkins':
    log      => '/var/log/jenkins/jenkins.log',
    options => [
      'daily',
      'copytruncate',
      'missingok',
      'rotate 7',
      'compress',
      'delaycompress',
      'notifempty'
    ],
  }
}

```

Designing convenient roles

There are several approaches to building roles, and you must decide which ones are most convenient for you and your team.

High-quality roles strike a balance between readability and maintainability. For most people, the benefit of seeing the entire role in a single file outweighs the maintenance cost of repetition. Later, if you find the repetition burdensome, you can change your approach to reduce it. This might involve combining several similar roles into a more complex role, creating sub-roles that other roles can include, or pushing more complexity into your profiles.

So, begin with granular roles and deviate from them only in small, carefully considered steps.

Here's the basic Jenkins role we're starting with:

```

class role::jenkins::master {
  include profile::base
  include profile::server
  include profile::jenkins::master
}

```

Related information

[Rules for role classes](#) on page 644

There are rules for writing role classes.

First approach: Granular roles

The simplest approach is to make one role per type of node, period. For example, the Puppet Release Engineering (RE) team manages some additional resources on their Jenkins masters.

With granular roles, we'd have at least two Jenkins master roles. A basic one:

```
class role::jenkins::master {
  include profile::base
  include profile::server
  include profile::jenkins::master
}
```

...and an RE-specific one:

```
class role::jenkins::master::release {
  include profile::base
  include profile::server
  include profile::jenkins::master
  include profile::jenkins::master::release
}
```

The benefits of this setup are:

- Readability — By looking at a single class, you can immediately see which profiles make up each type of node.
- Simplicity — Each role is just a linear list of profiles.

Some drawbacks are:

- Role bloat — If you have a lot of only-slightly-different nodes, you quickly have a large number of roles.
- Repetition — The two roles above are almost identical, with one difference. If they're two separate roles, it's harder to see how they're related to each other, and updating them can be more annoying.

Second approach: Conditional logic

Alternatively, you can use conditional logic to handle differences between closely-related kinds of nodes.

```
class role::jenkins::master::release {
  include profile::base
  include profile::server
  include profile::jenkins::master

  if $facts['group'] == 'release' {
    include profile::jenkins::master::release
  }
}
```

The benefits of this approach are:

- You have fewer roles, and they're easy to maintain.

The drawbacks are:

- Reduced readability...maybe. Conditional logic isn't usually hard to read, especially in a simple case like this, but you might feel tempted to add a bunch of new custom facts to accommodate complex roles. This can make roles much harder to read, because a reader must also know what those facts mean.

In short, be careful of turning your node classification system inside-out. You might have a better time if you separate the roles and assign them with your node classifier.

Third approach: Nested roles

Another way of reducing repetition is to let roles include other roles.

```
class role::jenkins::master {
  # Parent role:
  include role::server
  # Unique classes:
  include profile::jenkins::master
}

class role::jenkins::master::release {
  # Parent role:
  include role::jenkins::master
  # Unique classes:
  include profile::jenkins::master::release
}
```

In this example, we reduce boilerplate by having `role::jenkins::master` include `role::server`. When `role::jenkins::master::release` includes `role::jenkins::master`, it automatically gets `role::server` as well. With this approach, any given role only needs to:

- Include the "parent" role that it most resembles.
- Include the small handful of classes that differentiate it from its parent.

The benefits of this approach are:

- You have fewer roles, and they're easy to maintain.
- Increased visibility in your node classifier.

The drawbacks are:

- Reduced readability: You have to open more files to see the real content of a role. This isn't much of a problem if you go only one level deep, but it can get cumbersome around three or four.

Fourth approach: Multiple roles per node

In general, we recommend that you assign only one role to a node. In an infrastructure where nodes usually provide one primary service, that's the best way to work.

However, if your nodes tend to provide more than one primary service, it can make sense to assign multiple roles.

For example, say you have a large application that is usually composed of an application server, a database server, and a web server. To enable lighter-weight testing during development, you've decided to provide an "all-in-one" node type to your developers. You could do this by creating a new `role::our_application::monolithic` class, which includes all of the profiles that compose the three normal roles, but you might find it simpler to use your node classifier to assign all three roles (`role::our_application::app`, `role::our_application::db`, and `role::our_application::web`) to those all-in-one machines.

The benefit of this approach are:

- You have fewer roles, and they're easy to maintain.

The drawbacks are:

- There's no actual "role" that describes your multi-purpose nodes; instead, the source of truth for what's on them is spread out between your roles and your node classifier, and you must cross-reference to understand their configurations. This reduces readability.
- The normal and all-in-one versions of a complex application are likely to have other subtle differences you need to account for, which might mean making your "normal" roles more complex. It's possible that making a separate role for this kind of node would *reduce* your overall complexity, even though it increases the number of roles and adds repetition.

Fifth approach: Super profiles

Because profiles can already include other profiles, you can decide to enforce an additional rule at your business: all profiles must include any other profiles needed to manage a complete node that provides that service.

For example, our `profile::jenkins::master` class could include both `profile::server` and `profile::base`, and you could manage a Jenkins master server by directly assigning `profile::jenkins::master` in your node classifier. In other words, a "main" profile would do all the work that a role usually does, and the roles layer would no longer be necessary.

The benefits of this approach are:

- The chain of dependencies for a complex service can be more clear this way.
- Depending on how you conceptualize code, this can be easier in a lot of ways!

The drawbacks are:

- Loss of flexibility. This reduces the number of ways in which your roles can be combined, and reduces your ability to use alternate implementations of dependencies for nodes with different requirements.
- Reduced readability, on a much grander scale. Like with nested roles, you lose the advantage of a clean, straightforward list of what a node consists of. Unlike nested roles, you also lose the clear division between "top-level" complete system configurations (roles) and "mid-level" groupings of technologies (profiles). Not every profile makes sense as an entire system, so you some way to keep track of which profiles are the top-level ones.

Some people really find continuous hierarchies easier to reason about than sharply divided layers. If everyone in your organization is on the same page about this, a "profiles and profiles" approach might make sense. But we strongly caution you against it unless you're very sure; for most people, a true roles and profiles approach works better. Try the well-traveled path first.

Sixth approach: Building roles in the node classifier

Instead of building roles with the Puppet language and then assigning them to nodes with your node classifier, you might find your classifier flexible enough to build roles directly.

For example, you might create a "Jenkins masters" group in the console and assign it the `profile::base`, `profile::server`, and `profile::jenkins::master` classes, doing much the same job as our basic `role::jenkins::master` class.

Important:

If you're doing this, make sure you don't set parameters for profiles in the classifier. Continue to use Hiera / Puppet lookup to configure profiles.

This is because profiles are allowed to include other profiles, which interacts badly with the resource-like behavior that node classifiers use to set class parameters.

The benefits of this approach are:

- Your node classifier becomes much more powerful, and can be a central point of collaboration for managing nodes.
- Increased readability: A node's page in the console displays the full content of its role, without having to cross-reference with manifests in your `role` module.

The drawbacks are:

- Loss of flexibility. The Puppet language's conditional logic is often more flexible and convenient than most node classifiers, including the console.
- Your roles are no longer in the same code repository as your profiles, and it's more difficult to make them follow the same code promotion processes.

Puppet Forge

Puppet Forge is a collection of modules and how-to guides developed by Puppet and its community.

Modules manage a specific technology in your infrastructure and serve as the basic building blocks of Puppet desired state management. On the Puppet Forge, there is a module to manage almost any part of your infrastructure. Whether you want to manage packages or patch operating systems, a module is already set up for you. See each module's README for installation instructions, usage, and code examples.

When using an existing module from the Forge, most of the Puppet code is written for you. You just need to install the module and its dependencies and write a small amount of code (known as a profile) to tie things together. Take a look at our [Getting started with PE guide](#) to see an example of writing a profile for an existing module. For more information about existing modules, see the [module fundamentals documentation](#) and [Puppet Forge](#).

Puppet Development Kit (PDK)

You can write your own Puppet code and modules using Puppet Development Kit (PDK), which is a framework to successfully build, test and validate your modules.

Note that most Puppet users won't have to write full Puppet code at all, though you can if you want to. For installation instructions and more information, see the [PDK documentation](#).

Puppet VSCode extension

Puppet has an extension for Visual Studio Code (VSCode) — Microsoft's cross-platform source-code editor.

The *Puppet VSCode extension* makes writing and managing Puppet code easier and ensures your code is high quality. Its features include Puppet DSL intellisense, linting, and built-in commands. You can use the extension with Windows, Linux, or macOS. For installation instructions and a full list of features, see the [Puppet VSCode extension documentation](#).

Bolt

Bolt is an open source orchestration tool. You can use Bolt as a stand-alone tool or integrate it with Puppet.

Bolt allows you to automate tasks on an as-needed basis or as part of a greater orchestration workflow. To get started, see the [Bolt documentation](#).

Example configurations

Try out some common configuration tasks to see how you can use Puppet to manage your IT infrastructure.

- [Manage an NTP service](#) on page 669

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

- [Manage sudo privileges](#) on page 672

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

- [Manage a DNS nameserver file](#) on page 675

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

- [Manage firewall rules](#) on page 678

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

- [Forge examples](#) on page 681

Puppet Forge is a collection of modules and how-to guides developed by Puppet and its community.

Manage an NTP service

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

Before you begin

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

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

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

Using the Puppet NTP module, you can:

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

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

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

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

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

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

The resulting output is similar to this:

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

That's it! You've just installed the `puppetlabs-ntp` module.

2. The next step is adding classes from the NTP module to the main manifest.

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

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

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

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

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

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

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

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

Note: For additional time server options, see the list at <https://www.ntppool.org/>.

- d) On your agent, start a Puppet run:

```
puppet agent -t
```

Your Puppet-managed node is now configured to use NTP.

3. To check if the NTP service is running, run:

```
puppet resource service ntpd
```

On Ubuntu operating systems, the service is `ntp` instead of `ntpd`.

The result looks like this:

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

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

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

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

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

The `site.pp` looks like this:

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

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

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

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

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

Puppet offers many opportunities for learning and training, from formal certification courses to guided online lessons. See the [Learning Puppet page](#) for more information.

Manage sudo privileges

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

Before you begin

Before starting this walk-through, complete [the previous exercises](#).

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

Using this guide, you learn how to:

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

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

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

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

The resulting output is similar to this:

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

That's it! You've installed the `saz-sudo` module.

2. Next, you'll create a module that contains the `privileges` class.

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

- a) From the command line on the primary server, navigate to the modules directory:

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

- b) Create the module directory and its manifests directory:

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

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

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

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

- d) Save and exit the file.

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

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

- a) From the command line on the primary server, navigate to the main manifest:

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

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

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

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

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

- c) Save and exit the file.
d) On your primary server, ensure that there are no errors:

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

The parser returns nothing if there are no errors.

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

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

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

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

The results resemble the following:

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

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

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

Puppet offers many opportunities for learning and training, from formal certification courses to guided online lessons. See the [Learning Puppet page](#) for more information.

Manage a DNS nameserver file

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

Before you begin

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

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

In this exercise, you learn how to:

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

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

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

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

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

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

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

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

- a) From the command line on the Puppet primary server, navigate to the modules directory:

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

- b) Create the module directory and its templates directory:

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

- c) Use your text editor to create a file called `resolv.conf.erb` inside the `resolver/templates` directory.
- d) Edit the `resolv.conf.erb` file to add the following Ruby code:

```
# Resolv.conf generated by Puppet

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

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

- e) Save and exit the file.

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

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

- a) On the primary server, open `/etc/resolv.conf` with your text editor, and copy the IP address of your primary server's nameserver. In this example, the nameserver is `10.0.2.3`.
- b) Navigate to the main manifest:

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

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

```
$nameservers = ['10.0.2.3']

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

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

To see the results in the `resolv.conf` file, run:

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

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

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

Note the following about your new class:

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

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

- a) On the agent to which you applied the `resolver` class, edit `/etc/resolv.conf` to contain any nameserver IP address other than the one you want to use.
- b) Save and exit the file.
- c) Now, fix the mistake you've introduced. From the command line on your agent, run: `puppet agent -t --onetime`

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

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

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

For more information about working with Puppet and DNS, see our [Dealing with Name Resolution Issues](#) blog post.

Puppet offers many opportunities for learning and training, from formal certification courses to guided online lessons. See the [Learning Puppet page](#) for more information.

Manage firewall rules

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

Before you begin

Before starting this walk-through, complete the previous exercises in the [common configuration tasks](#).

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

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

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

Using this guide, you learn how to:

- Install the `puppetlabs-firewall` module.
 - Write a module to define the firewall rules for your Puppet managed infrastructure.
 - Add the firewall module to the main manifest.
 - Enforce the desired state using the `my_firewall` class.
1. The first step is installing the `puppetlabs-firewall` module from the Puppet Forge. The module introduces the `firewall` resource, which is used to manage and configure firewall rules. For more information about the `puppetlabs-firewall` module, see its [README](#). To install the module, on the primary server, run:

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

The resulting output is similar to this:

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

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

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

- a) From the command line on the primary server, navigate to the modules directory:

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

- b) Create the module directory and its manifests directory:

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

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

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

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

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

- e) Save and exit the file.
- f) From the `manifests` directory, use your text editor to create `post.pp`.
- g) The post rules tell the firewall to drop requests that haven't met the rules defined by `pre.pp` or in `site.pp`. Edit `post.pp` so it contains the following Puppet code:

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

- h) Save and exit the file.

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

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

- a) On the primary server, navigate to the main manifest:

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

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

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

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

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

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

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

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

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

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

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

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

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

The resulting output is similar to this:

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

Chain FORWARD (policy ACCEPT)
target     prot opt source                destination

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


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

- a) On an agent where you applied the `my_firewall` class, run:

```
iptables --list
```

Note that the rules from the `my_firewall` class have been applied.

- b) From the command line, add a rule to allow connections to port 8449 by running:

```
iptables -I INPUT -m state --state NEW -m tcp -p tcp --dport 8449 -j
ACCEPT
```

- c) Run `iptables --list` again and see that this new rule is now listed.

- d) Run Puppet on that agent:

```
puppet agent -t --onetime
```

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

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

You can learn more about the Puppet firewall module by visiting the [Forge](#).

Puppet offers many opportunities for learning and training, from formal certification courses to guided online lessons. See the [Learning Puppet page](#) for more information.

Forge examples

Puppet Forge is a collection of modules and how-to guides developed by Puppet and its community.

The [Forge](#) has existing modules and code examples that assist with automating the following use cases:

- **Base system configuration**
 - Including [registry](#), [NTP](#), [firewalls](#), [services](#)
- **Manage web servers**
 - Including [apache](#), [tomcat](#), [IIS](#), [nginx](#)
- **Manage database systems**
 - Including [Oracle](#), [Microsoft SQL Server](#), [MySQL](#), [PostgreSQL](#)
- **Manage middleware/application systems**
 - Including [Java](#), [WebLogic/Fusion](#), [IBM MQ](#), [IBM IIB](#), [RabbitMQ](#), [ActiveMQ](#), [Redis](#), [ElasticSearch](#)
- **Source control**
 - Including [Github](#), [Gitlab](#)
- **Monitoring**
 - Including [Splunk](#), [Nagios](#), [Zabbix](#), [Sensu](#), [Prometheus](#), [NewRelic](#), [Icinga](#), [SNMP](#)
- **Patch management**
 - [OS patching](#) on Enterprise Linux, Debian, SLES, Ubuntu, Windows
- **Package management**
 - Linux: Puppet integrates directly with native package managers
 - Windows: Use Puppet to install software directly on Windows, or integrate with [Chocolatey](#)
- **Containers and cloud native**
 - Including [Docker](#), [Kubernetes](#), [Terraform](#), [OpenShift](#)

- **Networking**
 - Including [Cisco Catalyst](#), [Cisco Nexus](#), [F5](#), [Palo Alto](#), [Barracuda](#)
- **Secrets management**
 - Including [Hashicorp Vault](#), [CyberArk Conjur](#), [Azure Key Vault](#), [Consul Data](#)

See each module's README for installation, usage, and code examples.

References

Puppet references, including configuration settings, functions, and metaparameters.

- [Experimental features](#) on page 682

Released versions of Puppet can include experimental features to be considered for adoption but are not yet ready for production. These features need to be tested in the field before they can be considered safe, and therefore are turned off by default.

- [Configuration settings reference](#)
- [Metaparameter reference](#) on page 684

Metaparameters are attributes that work with any resource type, including custom types and defined types. They change the way Puppet handles resources.

- [Built-in functions](#)
- [Ruby API for developing extensions](#)
- [Puppet CLI](#)

Experimental features

Released versions of Puppet can include experimental features to be considered for adoption but are not yet ready for production. These features need to be tested in the field before they can be considered safe, and therefore are turned off by default.

Experimental features can have a solid design but with an unknown performance and resource usage. Sometimes even the design is tentative, and because of this, we need feedback from users. By shipping these features early in disabled form, we want it to be easier for testing and giving feedback.

Risks and support



CAUTION: Experimental features are not officially supported by Puppet, and we do not recommend that you turn them on in a production environment. They are available for testing in relatively safe scratch environments, and are used at your own risk.

Puppet employees and community members do their best to help you in informal channels like IRC, and the puppet-users and puppet-dev mailing lists, but we make no promises about experimental functionality.

Enabling experimental features might degrade the performance of your Puppet infrastructure, interfere with the normal operation of your managed nodes, introduce unexpected security risks, or have other undesired effects.

This is especially relevant to Puppet Enterprise customers. If Puppet support is assisting you with a problem, we might ask you to disable any experimental features.

Changes to experimental features

Experimental features are exempt from semantic versioning, which means that they can change at any time, and are not limited to major or minor release boundaries.

These changes might include adding or removing functionality, changing the names of settings and other affordances, and more.

Documentation of experimental features

The Puppet documentation contains pages for currently available experimental features. These pages are focused on enabling a feature and running through the interesting parts of its functionality; they might lag slightly behind the feature as implemented.

When a feature has experienced major changes across minor versions, we note the differences at the top of that feature page.

Each feature page attempts to give some context about the status of that feature and its prospects for official release.

Giving feedback on experimental features

To help us keep improving Puppet, tell us more about your experience.

The best places to talk about experimental features are the [puppet-users](#) and [puppet-dev](#) mailing lists. This tells us what's working and what isn't, while also helping others learn from your experience. For more information about the Puppet mailing lists, see the [community guidelines](#) for mailing lists.

For more immediate conversations, you can use the #puppet and #puppet-dev channels on irc.freenode.net. For more information about these channels, see the [community guidelines](#) for IRC.

- [Msgpack support](#) on page 683

Puppet agents and primary servers communicate over HTTPS, exchanging structured data in JSON, or PSON which allows binary data.

Msgpack support

Puppet agents and primary servers communicate over HTTPS, exchanging structured data in JSON, or PSON which allows binary data.

[Msgpack](#) is an efficient serialization protocol that behaves similarly to JSON. It provides faster and more robust serialization for agent-server communications, without requiring many changes in our code.

Important: When msgpack is enabled, the primary Puppet server and agent communicates using msgpack instead of PSON.

Enabling Msgpack serialization

Enabling msgpack is easy, but first, it must be installed because the gem is not included in the puppet-agent or puppetserver packages.

1. Install the [msgpack gem](#) on your primary server and all agent nodes.

If you are using the Puppet Enterprise test environment, make sure to use PE gem command instead of the system gem command.

On *nix nodes, run the following command:

```
/opt/puppetlabs/puppet/bin/gem install msgpack
```

On Windows nodes, run the following command:

```
"C:\Program Files\Puppet Labs\Puppet\sys\ruby\bin\gem" install msgpack
```

On Puppet Server, run the following command and then restart the Puppet Server service:

```
puppetserver gem install msgpack
```

2. In the [agent] or [main] section of puppet.conf on any number of agent nodes, set the [preferred_serialization_format](#) setting to msgpack.

After this is configured, the primary Puppet server uses msgpack when serving any agents that have `preferred_serialization_format` set to msgpack. Any agents without that setting continue to receive PSON as normal.

Metaparameter reference

Metaparameters are attributes that work with any resource type, including custom types and defined types. They change the way Puppet handles resources.

With metaparameters, you can change how Puppet handles specific resources. For example, you can:

- Add metadata to a resource with the `alias` or `tag` metaparameters.
- Set limits on when the resource should be applied, by using relationship metaparameters like `notify` or `require`.
- Prevent Puppet from making changes, by setting the `noop` metaparameter.
- Change logging verbosity with the `loglevel` metaparameter.

alias

Creates an alias for the resource. You can explicitly specify the `alias` metaparameter, but it's usually safer to give the resource the `alias` value as the title and provide the full `namevar` value explicitly.

For example, this sample gives the title as `sshdconfig`, which acts as the alias. The `namevar` value is the path, which is set to `'/etc/ssh/sshd_config'`.

```
file { 'sshdconfig':
  path => '/etc/ssh/sshd_config',
  source => '...'
}

service { 'sshd':
  subscribe => File['sshdconfig'],
}
```

Aliases generally work only for creating relationships; anything else that refers to an existing resource (such as amending or overriding resource attributes in an inherited class) must use the resource's exact title. For example, the following code will not work, because there's no way for Puppet to know that the two stanzas should affect the same file.

```
file { '/etc/ssh/sshd_config':
  owner => root,
  group => root,
  alias => 'sshdconfig',
}

File['sshdconfig'] {
  mode => '0644',
}
```

audit

Marks a subset of this resource's unmanaged attributes for auditing. Accepts an attribute name, an array of attribute names, or the value `all`.

When `audit` is set for an attribute, when Puppet applies the catalog, it checks whether that attribute of the resource has been modified, comparing its current value to the previous run. Any change is then logged alongside any actions Puppet performed while applying the catalog.

before

Specifies one or more resources that depend on this resource, expressed as resource references. Specify multiple resources as an array of references. When specified, `before` causes the resource to be applied before the dependent resources. This is one of the four relationship metaparameters, along with `require`, `notify`, and `subscribe`. For more information about creating relationships between resources, see [Relationships and ordering](#) on page 405. For details about resource references, see [Resource and class references](#) on page 535.

loglevel

Sets the level at which information is logged. The log levels have the biggest impact when logs are sent to syslog, which is the default log. Any of the log levels are valid values for this metaparameter. The order of the log levels, in decreasing priority, is:

- emerg
- alert
- crit
- err
- warning
- notice
- info
- verbose
- debug

noop

Whether to apply this resource in non-operational, or "no-op" mode. When applying a resource in `noop` mode, Puppet checks whether the resource is in the desired state as declared in the catalog. If the resource is not in the desired state, Puppet takes no action, but reports the changes it would have made. These simulated changes appear in the report sent to the primary server or are displayed be shown on the console if running puppet agent or puppet apply in the foreground. The simulated changes do not send refresh events to any subscribing or notified resources, although Puppet logs that a refresh event would have been sent. Valid values are `true` or `false`.

Note: The `noop` setting allows you to globally enable or disable `noop` mode, but it does not override the `noop` metaparameter on individual resources. That is, the value of a global `noop` setting affects only resources that do not have an explicit value set for their `noop` attribute.

notify

Specifies one or more resources that depend on this resource, expressed as resource references. Specify multiple resources as an array of references.

When this attribute is set, this resource is applied before the notified resources. If Puppet makes changes to this resource, it causes all of the notified resources to refresh. Refresh behavior varies by resource type: for example, services restart and mounts unmount and re-mount. Not all types can refresh.

This is one of the four relationship metaparameters, along with `before`, `require`, and `subscribe`. For more information about creating relationships between resources, see [Relationships and ordering](#) on page 405. For details about resource references, see [Resource and class references](#) on page 535.

require

Specifies one or more resources that depend on this resource, expressed as resource references. Specify multiple resources as an array of references.

When this attribute is set, the required resources are applied before this resource.

This is one of the four relationship metaparameters, along with `before`, `notify`, and `subscribe`. For more information about creating relationships between resources, see [Relationships and ordering](#) on page 405. For details about resource references, see [Resource and class references](#) on page 535.

schedule

A schedule to govern when Puppet is allowed to manage this resource. The value of this metaparameter must be the name of a schedule resource. This means you must first declare a schedule resource, then refer to it by name.

For example:

```
schedule { 'everyday':
  period => daily,
  range  => "2-4"
}

exec { ["/usr/bin/apt-get update"]:
  schedule => 'everyday'
}
```

You can declare the schedule resource anywhere in your manifests, as long as it ends up in the final compiled catalog.

See the [schedule type](#) for more information.

stage

Which run stage this class should reside in. To assign a class to a different stage, you must:

- Declare the new stage as a `stage` resource See the [stage type](#) for details.
- Declare an order relationship between the new stage and the main stage.
- Use the resource-like syntax to declare the class, and set the `stage` metaparameter to the name of the desired stage.

Important: This metaparameter can only be used on classes, and only when declaring them with the resource-like syntax. It cannot be used on normal resources or on classes declared with `include`. By default, all classes are declared in the main stage.

For example:

```
stage { 'pre':
  before => Stage['main'],
}

class { 'apt-updates':
  stage => 'pre',
}
```

subscribe

Specifies one or more resources that depend on this resource, expressed as resource references. Specify multiple resources as an array of references.

When this attribute is present, the subscribed resources are applied before this resource. If Puppet makes changes to any of the subscribed resources, it causes this resource to refresh. Refresh behavior varies by resource type: for example, services restart and mounts unmount and re-mount. Not all types can refresh.

This is one of the four relationship metaparameters, along with `before`, `require`, and `notify`. For more information about creating relationships between resources, see [Relationships and ordering](#) on page 405. For details about resource references, see [Resource and class references](#) on page 535.

tag

Add the specified tags to the associated resource. Although all resources are automatically tagged with as much information as possible, such as with each class and definition containing the resource, it can be useful to add your own tags to a given resource. Multiple tags can be specified as an array:

```
file {'/etc/hosts':  
  ensure => file,  
  source => 'puppet:///modules/site/hosts',  
  mode   => '0644',  
  tag    => ['bootstrap', 'minimumrun', 'mediumrun'],  
}
```

Tags are useful for things like applying a subset of a host's configuration with the `tags` configuration setting, such as with `puppet agent --test --tags bootstrap`. See the configuration reference for more information about the [tags setting](#)